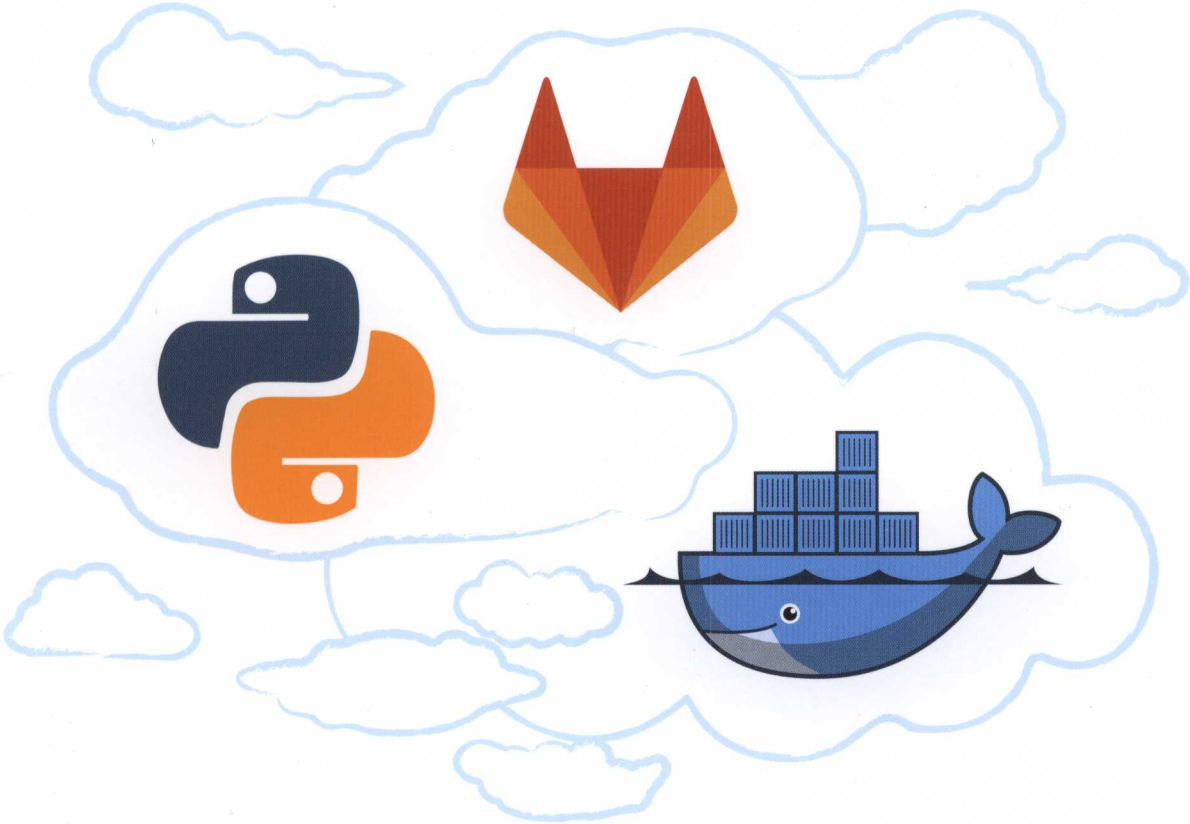
Broadview®
www.broadview.com.cn

图解方式全面讲解Python全栈实践流程
用Docker部署项目
用GitLab对Python项目持续集成



Python全栈开发实践入门

谢瑛俊◎编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

谢瑛俊

从事 Oracle 数据库性能优化维护、网站开发超过 10 年。目前担任企业开发工程师和数据分析师。巧遇 Python，为其简洁灵活所折服，潜心研究，将其应用于工作之中。现将自己在 Python 全栈开发中的多年经验与广大读者分享，借此互相学习交流，望有所参悟，修得正果。

Python全栈开发实践入门

谢瑛俊◎编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

全栈工程师不应只会前后端开发，而是应该从开发、测试、部署各个方面都有所掌握的全技能人才。本书使用了热门的 Docker 容器技术、GitLab 版本控制、GitLab-runner 持续集成、Python Web Flask 框架等，将一整套开发流程通过简单的案例展现出来。

本书适合想从事 IT 行业或刚刚毕业的新人们，通过阅读本书的案例可初步了解开发流程，本书也可作为各大院校相关专业的参考用书和相关培训机构的培训教材。

图书在版编目（CIP）数据

Python 全栈开发实践入门 / 谢瑛俊编著. —北京：电子工业出版社，2017.11
ISBN 978-7-121-32811-4

Ⅰ. ①P… Ⅱ. ①谢… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字（2017）第 242675 号

责任编辑：安　娜
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：17.75　字数：272 千字
版　　次：2017 年 11 月第 1 版
印　　次：2017 年 11 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

何为全栈工程师

全栈工程师是指掌握多种技能，并能利用多种技能独立完成产品的人，也叫全端工程师（同时具备前端和后台能力），其英文是 Full Stack Developer。[1] 而在教育体系中，人们常常告知你应该选择什么专业深造下去，在行业里做个专家。这是不是与全栈工程师冲突了呢？

我个人认为全栈工程师应该属于一个企业内 IT 部门的救火员，哪里需要他，他就出现在哪个岗位解决遇到的问题。他了解并掌握紧贴潮流的最新技术，能在某些特定领域提出自己的独特见解。

例如，在软考职称中，初级职称和中级职称分成了 5 个专业，每个专业又细分为多个不同的职称，初级和中级职称一共包含了 22 种职称。但到了高级职称后只剩下 5 种职称，每一种职称都是经过多种初级和中级职称融会贯通而成的，如下图所示。

1　全栈工程师解释引用
https://baike.baidu.com/item/%E5%85%A8%E6%A0%88%E5%B7%A5%E7%A8%8B%E5%B8%88/12983270?fr=aladdin

计算机技术与软件专业技术资格（水平）考试
专业类别、资格名称和级别层次对应表

	计算机软件	计算机网络	计算机应用技术	信息系统	信息服务
高级资格	信息系统项目管理师　系统分析师　系统架构设计师 网络规划设计师　系统规划与管理师				
中级资格	软件评测师 软件设计师 软件过程能力评估师	网络工程师	多媒体应用设计师 嵌入式系统设计师 计算机辅助设计师 电子商务设计师	系统集成项目管理工程师 信息系统监理师 信息安全工程师 数据库系统工程师 信息系统管理工程师	计算机硬件工程师 信息技术支持工程师
初级资格	程序员	网络管理员	多媒体应用制作技术员 电子商务技术员	信息系统运行管理员	网页制作员 信息处理技术员

软考职称对应表[1]

写书的目的

混混沌沌到了而立之年，工作 8 年时间里虽然写了很多的小程序，有很多实践，但是没有一项是能拿得出手来展示炫耀的，很多开发习惯更是不规范，随意性很大。

按照美国缅因州国家训练实验室的研究成果《学习金字塔》[2]中所描述，学习的最好效果就是把学会的知识传授给他人。

1　软件职称对应表：http://www.ruankao.org.cn/jsjnew/cms/focusExam/zgjs/

2　学习金字塔相关解释和图片：
http://baike.baidu.com/link?url=crFFN1DAv3tkXQYwRLoFJKKbSUEDxWUlQ4Qc3otYVLOQxYxSX37SUOQCQ6U0kM6_P4iev43a7v4BsBIvlcsVRKwTCK1L-VbyYsNbNEXmkBdlw1W1j73qT1qBvn_87pvDlNDHfoVCRCrVSGn8NmgXPa

学习金字塔

授人以鱼不如授人以渔，同时还可以把自己学到的知识内容认真梳理一遍。通过整理把知识点连贯起来，使其有较好的层次和顺序。

在网络上或者已出版的书籍中，还未看到有关利用 Docker 来开发 Python 项目并持续集成的完整开发流程的相关文章，因此本书提供了一些关于 Docker、Git、GitLab-runner、Flask Web 的简单案例，从服务器搭建、开发环境、代码写作、程序测试到持续集成一整套完整的开发流程。

关于语言之争

很多时候大家都会说 X 语言最好、最强，但是在我看来，语言只是一种工具。打个比方，C 可以看成美工刀（够锋利），Java 可以看成剪刀。裁纸的时候用美工刀的效率肯定高过剪刀，但是剪纸的时候用剪刀是不是比用美工刀快？美工刀不是不能剪纸，但是速度没剪刀来得快。

当你精通一门语言后，再学习其他语言的时候，学会的语言不会成为障碍，反而可以对这两种语言对比学习以加深理解。

前置知识

这里假设你懂点编程知识，本书的编程主要是以 Python 为主。[1]

1 免费课程 Python 完全零基础入门精讲 【优品课堂】：http://ute.ke.qq.com/

也假设你会用 Linux 命令，不会也没什么关系，我们会在用到的时候进行讲解，但是深入底层机制的知识只能靠你自己去查找资料来学习了，本书用到的操作系统主要以 Ubuntu 16.04.1 server amd64 LTS 为主。

关于作者

从大学到工作阶段经过 4 次考试才拿到中级职称的网络工程师证书，在学校的时候理论过关了，但是缺少实践，下午题答得不好；而工作之后，有设备有环境可以实践了，但又常常忘记理论知识。

在我们单位，IT 这块儿没有细分岗位，是一个大技术部兼管 IT 部门（设备维修和 IT 都同属于技术部），在部门内每个人都是技术能手。我平常工作大部分是围绕 DBA、服务器维护和数据分析这些方面。

曾经精通 C、VB、PHP、SQL，现在平时用得最多的是 Python，“人生苦短，我用 Python”。使用 Python 开发了几十个小程序，来解决工作中遇到的问题。

由于编者水平有限，书中疏漏之处在所难免，敬请谅解。

目　录

第1章
安装Ubuntu系统

1.1 普通镜像安装

1. 下载官方镜像

打开 https://www.ubuntu.com/download/server，选择下载所需镜像，这里下载的是 Ubuntu Server 16.04.1 LTS 镜像，如图 1-1 所示。

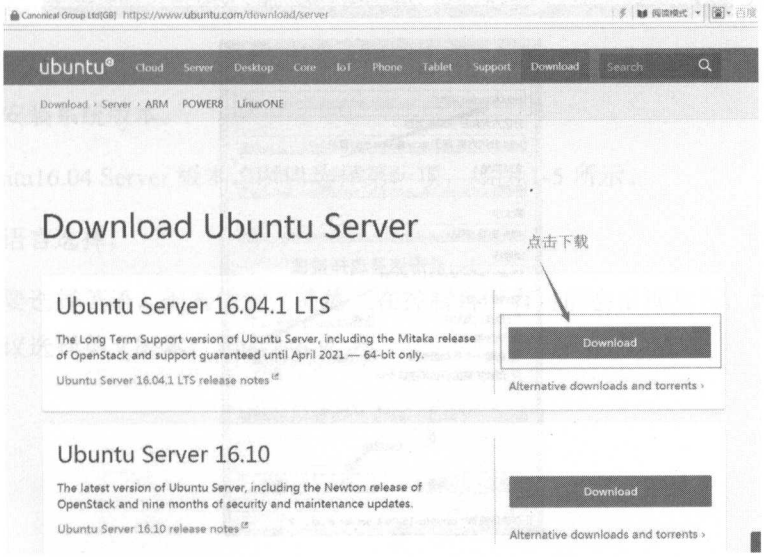


图 1-1

如果没有自动弹出镜像下载，则单击“download now”，如图 1-2 所示。

图 1-2

2. 制作启动 U 盘

将下载的 ISO 镜像刻录到光盘上或者通过 rufus 官网[1]将镜像解压到 U 盘上安装。图 1-3 为 rufus 制作启动 U 盘的方式。

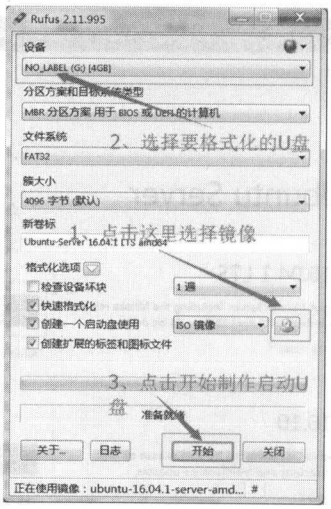


图 1-3

1 https://rufus.akeo.ie/?locale=zh_CN

3．开始安装

（1）选择语言。

一定要选择英语，如果选择中文（简体）则有可能出现“无法安装 busybox-initramfs”的问题，如图 1-4 所示。

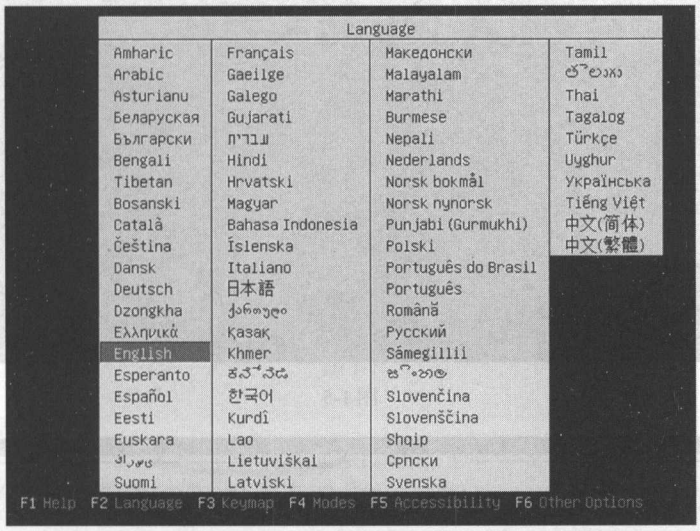


图 1-4

（2）选择安装系统版本。

安装 Ubuntu16.04 Server 版本，所以选择第一项，如图 1-5 所示。

（3）系统语言选择。

这里也需要选择英语，因为中文（简体）在控制台下有可能会出现乱码，为了避免这种情况的发生，建议选择英文版本，如图 1-6 所示。

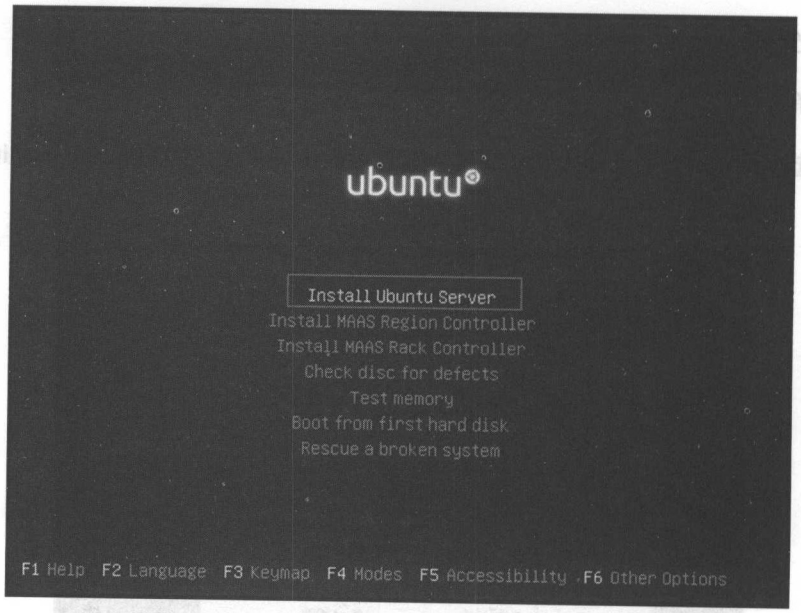


图 1-5

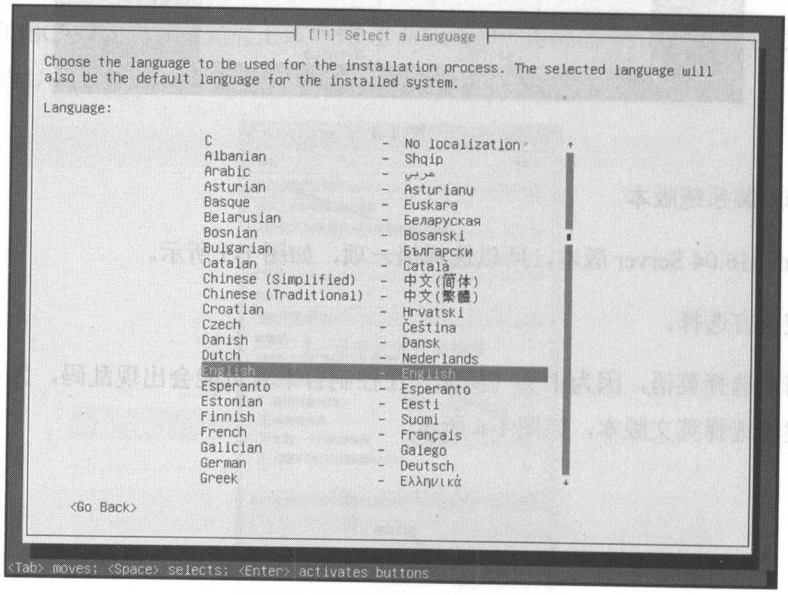


图 1-6

（4）所在地区选择，这里选择“Hong Kong”，如图 1-7 所示。

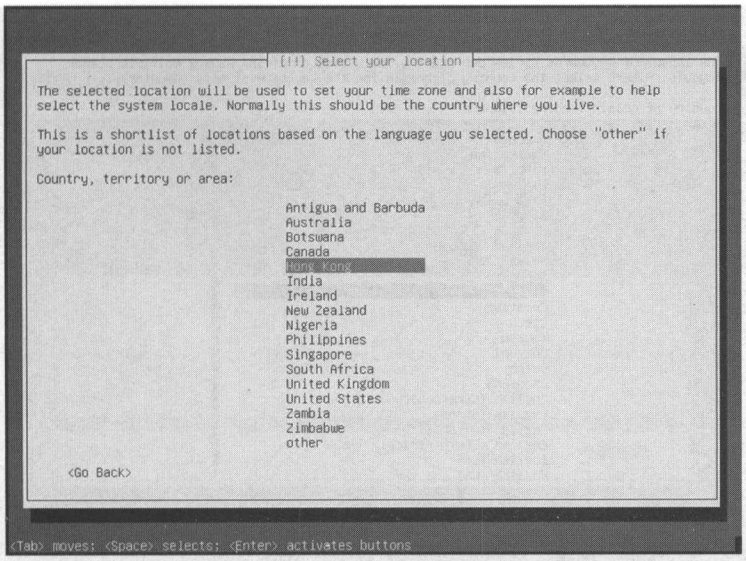


图 1-7

（5）键盘布局。

除特殊情况外一般不检测，如图 1-8 所示。

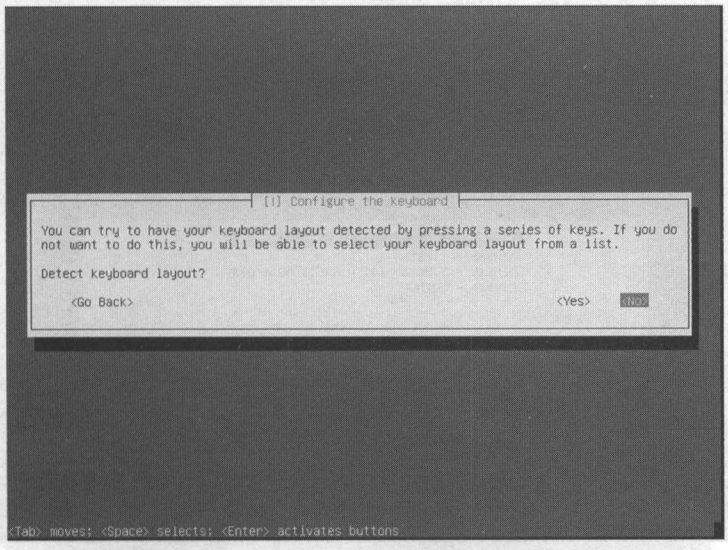


图 1-8

从键盘布局列表中选择键盘所属国家为：“Chinese”，如图 1-9 所示。

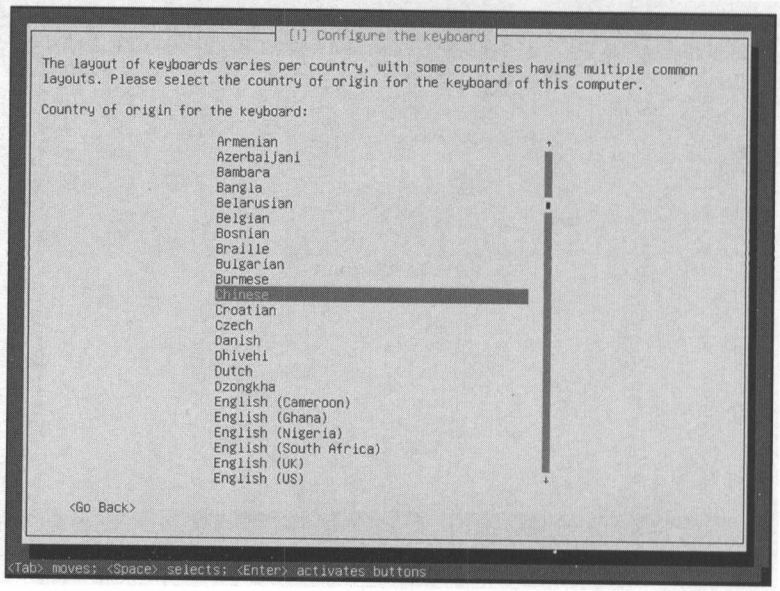


图 1-9

选择键盘布局方案，如图 1-10 所示。

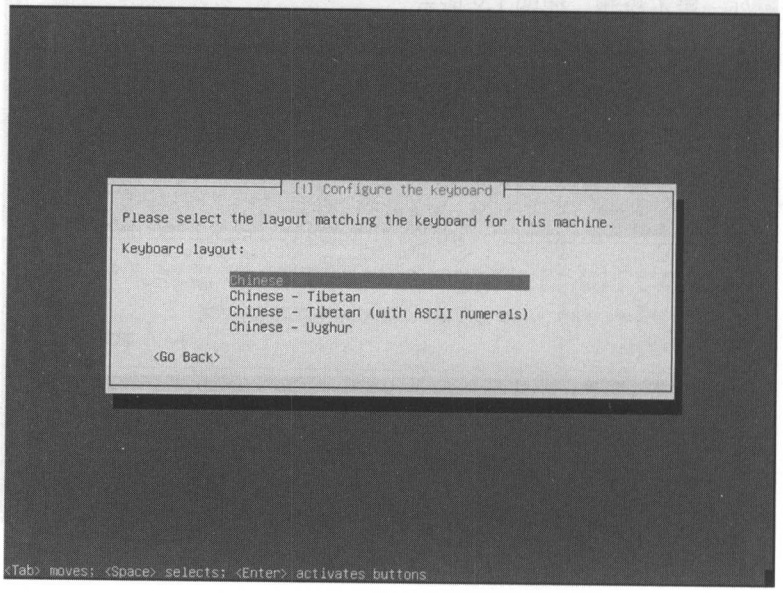


图 1-10

（6）设置服务器名称。

可以随便定义计算机名，但最好能从其名字就可以看出这台服务器的主要工作内容是什么，所以这里设置为“ubuntu-pxe-server”，如图 1-11 所示。

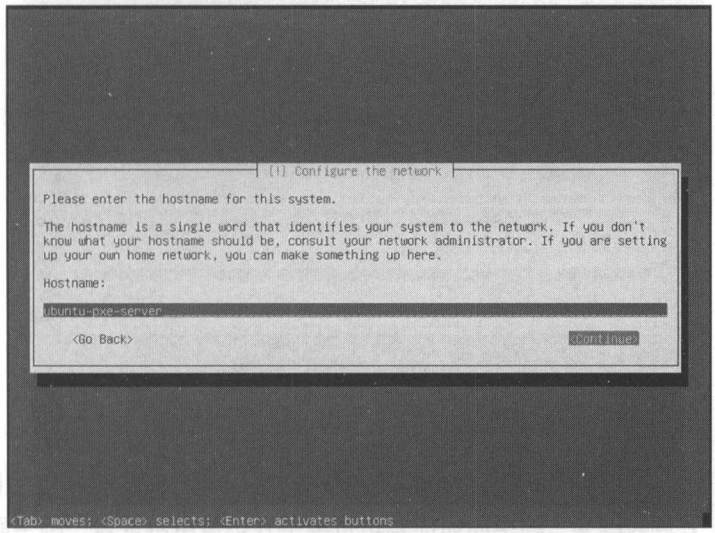


图 1-11

（7）设置管理员名称。

系统建议为全名，适用于发送邮件和定义软件所有者，如图 1-12 所示。

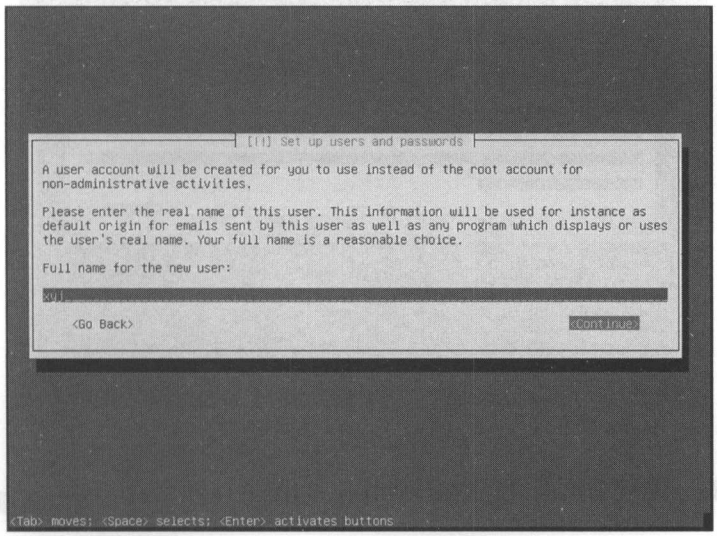


图 1-12

（8）设置登录账号。

默认是上一步设置的名字，也可以自定义，如图 1-13 所示。

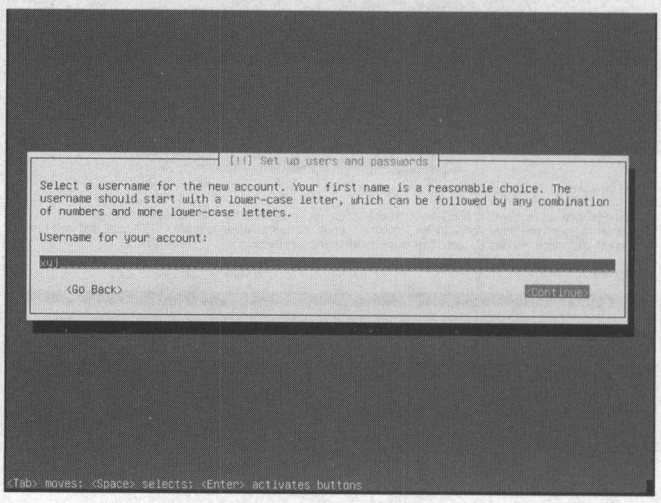


图 1-13

（9）设置登录密码，如图 1-14 所示。

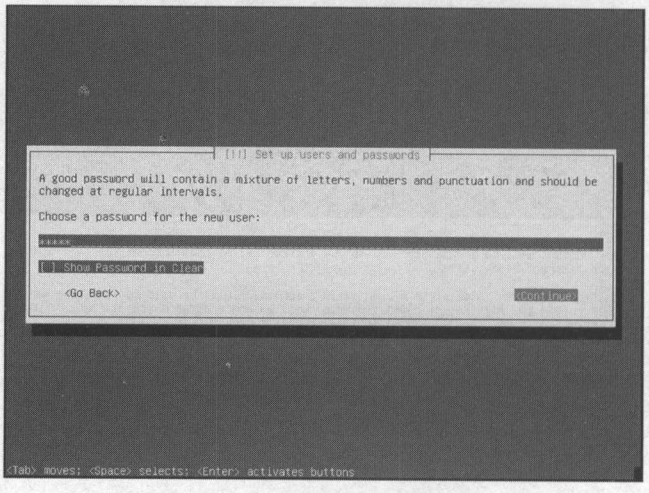


图 1-14

再次输入密码，需要与上面设置的密码相同，如图 1-15 所示。

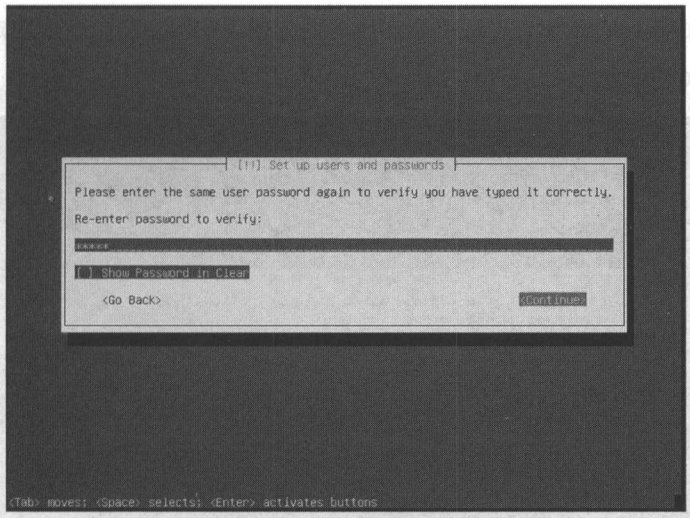


图 1-15

确认使用弱密码（有可能不提示，因为这里使用了全部小写字母的密码），弱密码提示如图 1-16 所示。强密码一般是指由最少八个字符组合而成的密码，密码内包含了大小写字母、数字和特殊符号。

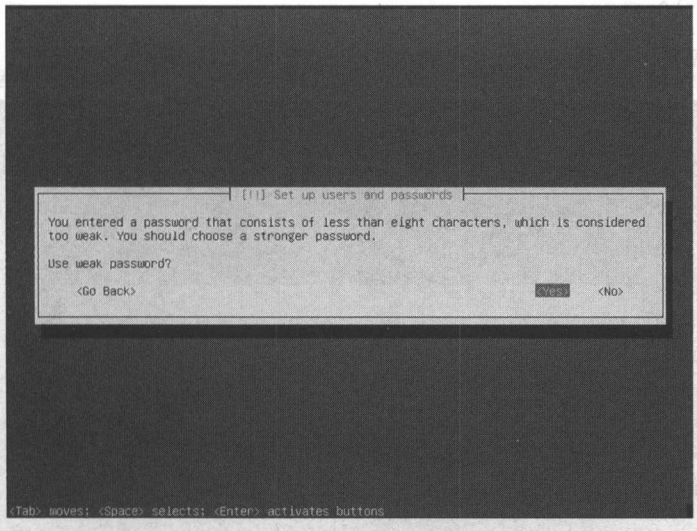


图 1-16

（10）选择是否加密主目录。

加密主目录如图 1-17 所示。千万不要在服务器上使用加密主目录，否则定时执行任务会无法执行，提示“not found”。

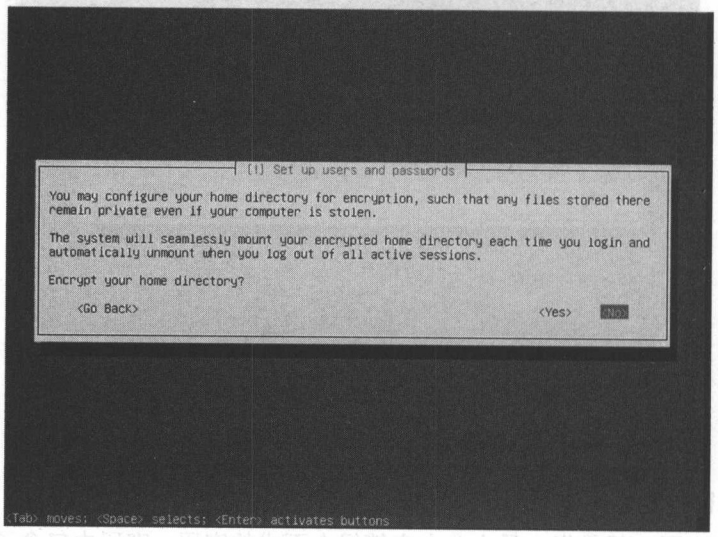


图 1-17

（11）设置时区。

一般会自动识别，这里识别到“亚洲/重庆”，如图 1-18 所示。

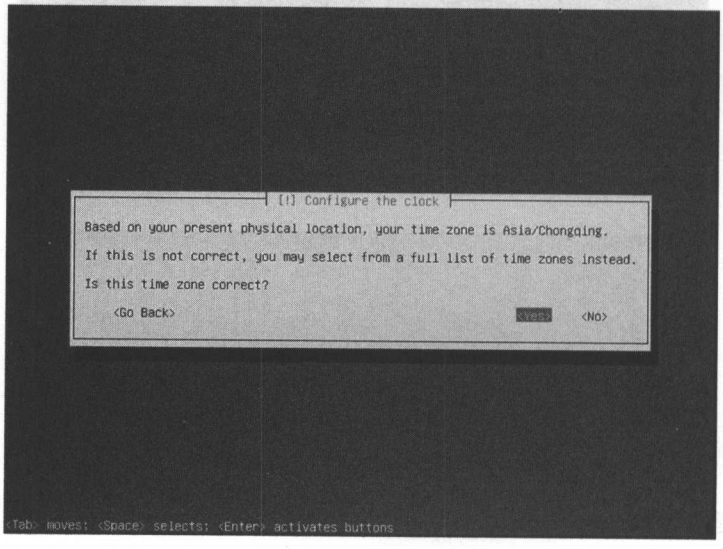


图 1-18

（12）硬盘分区。

选择使用整个硬盘并配置 LVM 分区，使用 LVM 分区的好处是可以忽视前期逻辑分区的大小划分，方便后期区域扩展（如果不是新硬盘，则应选择手动分区，不然会覆盖里面已有分区），如图 1-19 所示。

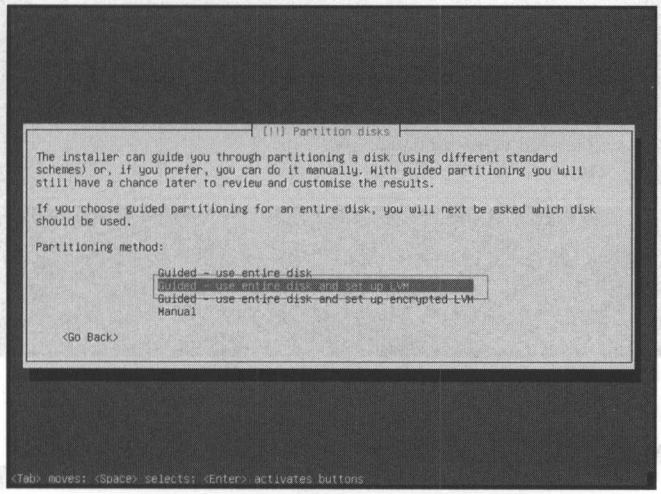


图 1-19

选择需要安装系统的硬盘，这里因为虚拟机只有一个硬盘，所以没有其他选项，如图 1-20 所示。

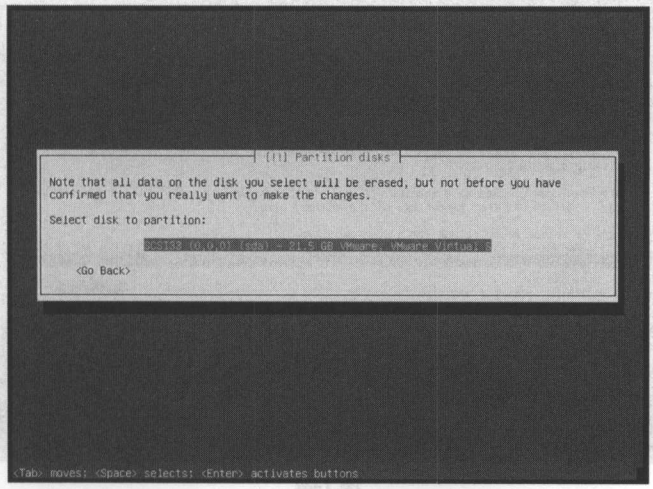


图 1-20

确认在选择好的硬盘上做分区表和配置 LVM，如图 1-21 所示。

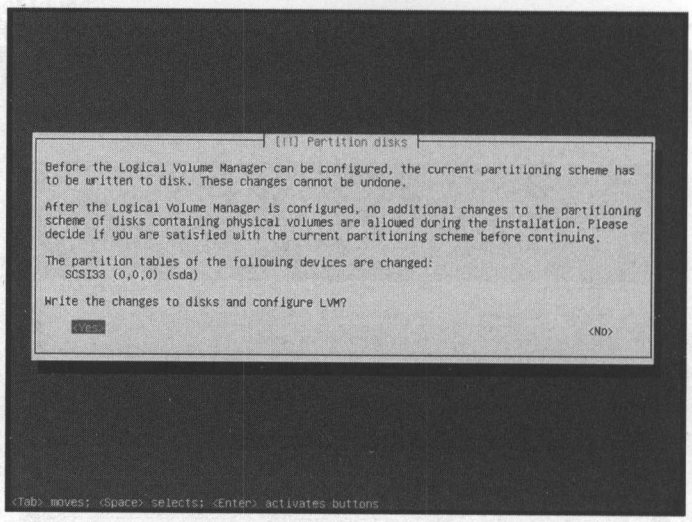


图 1-21

（13）配置 LVM。

选择需要配置 LVM 分区的大小，默认是全部空间，这里直接选择默认即可，如图 1-22 所示。

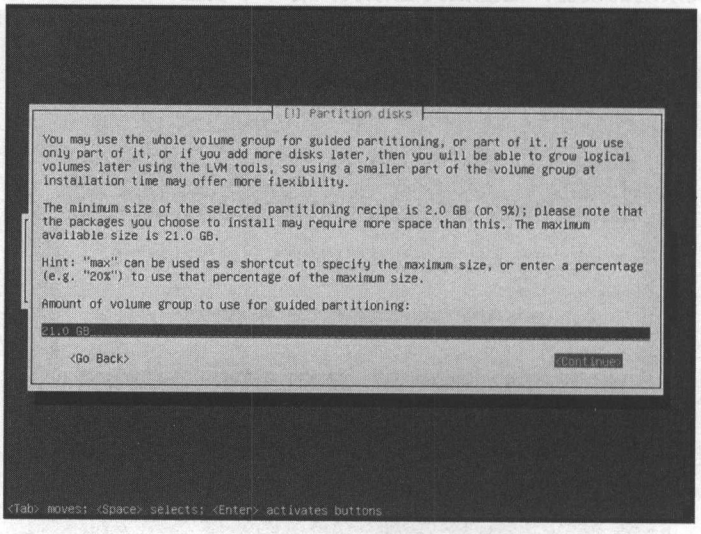


图 1-22

确认分区方案并写入硬盘中（需确定硬盘内没有其他数据，否则会导致原有分区表混乱，数据丢失），如图 1-23 所示。

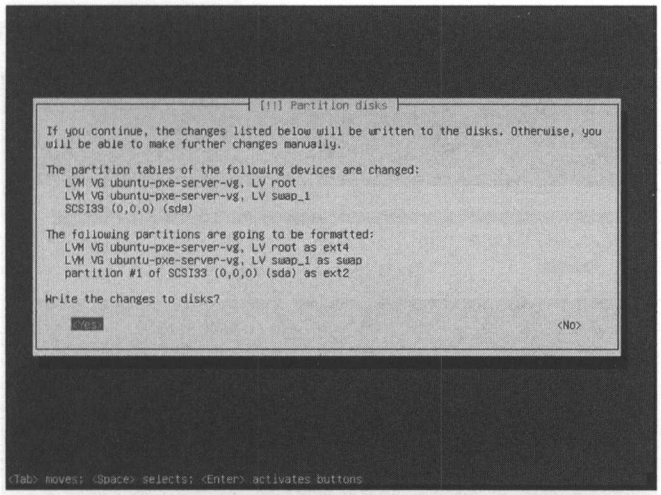


图 1-23

（14）代理配置。

一般直连网络能访问到互联网是不需要配置这项的，但有些特殊的网络环境会用到这个配置，这里忽略，如图 1-24 所示。

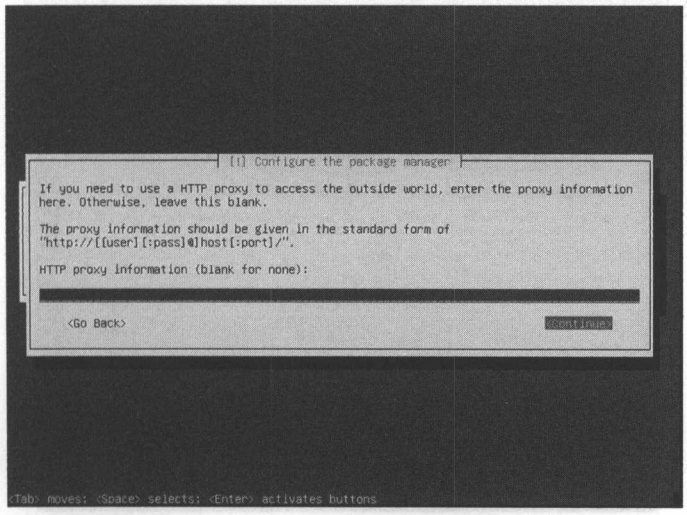


图 1-24

（15）更新数据源。

视网络环境状况下载，耐心等待，如图 1-25 所示。

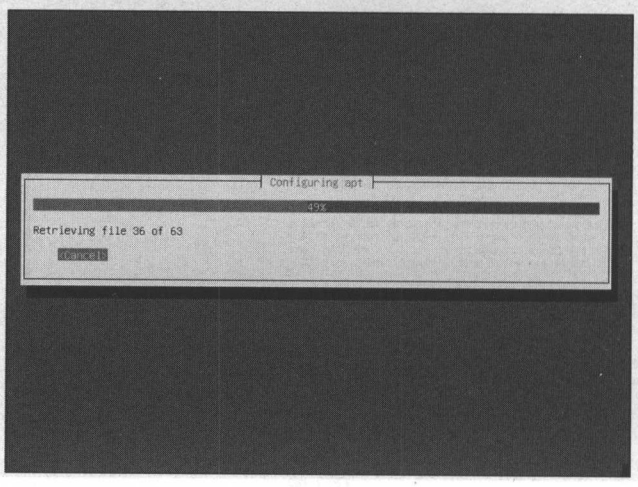


图 1-25

（16）配置是否自动更新补丁。

在测试服务器上可以随便设置，但是在生产环境下最好不要自动更新，以避免出现软件版本不兼容的情况。第一项是不自动更新，第二项是自动更新，第三项是使用 Landscape 管理套件控制，如图 1-26 所示。

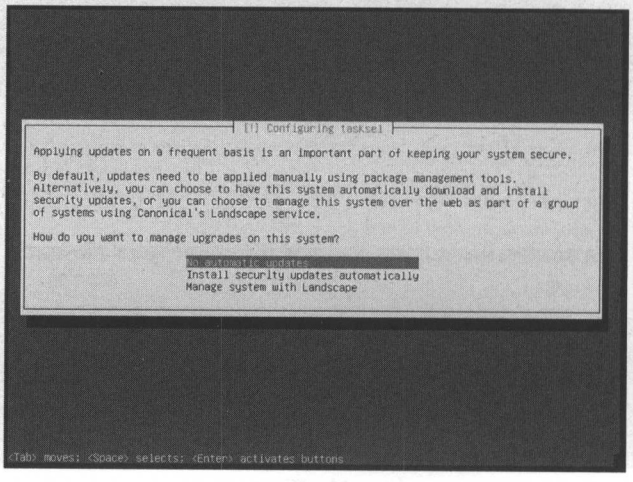


图 1-26

（17）选择需要安装的软件包。

这里不仅选择了默认的“standard system utilities”，还选择了“OpenSSH server”，这样在服务器安装好后就可以使用 SSH 控制了，如图 1-27 所示。

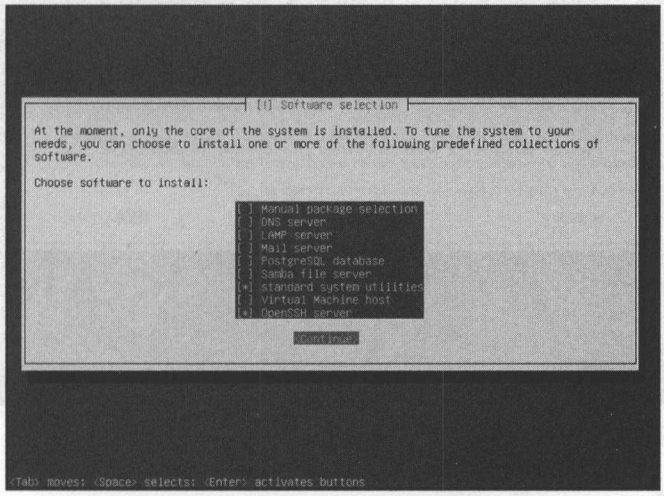


图 1-27

安装软件中，耐心等候，如图 1-28 所示。

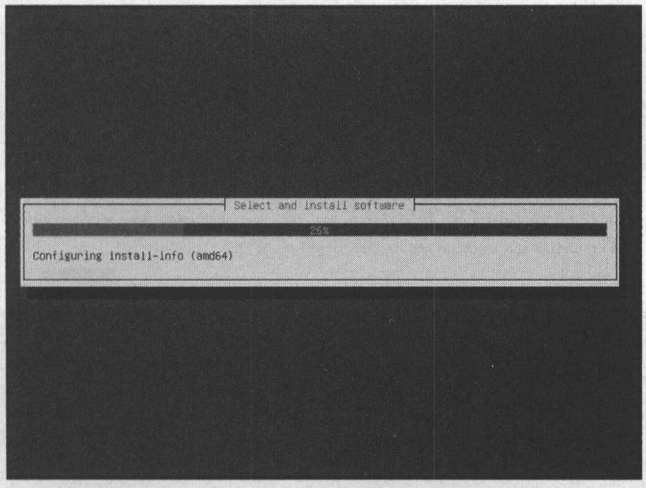


图 1-28

（18）配置引导程序。

安装引导程序到磁盘 boot 分区中，如图 1-29 所示。

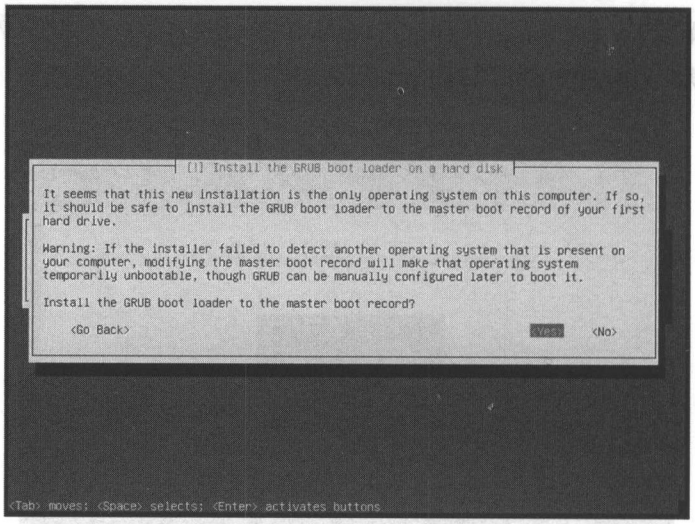


图 1-29

（19）安装完成。

重启系统即可，如图 1-30 所示。

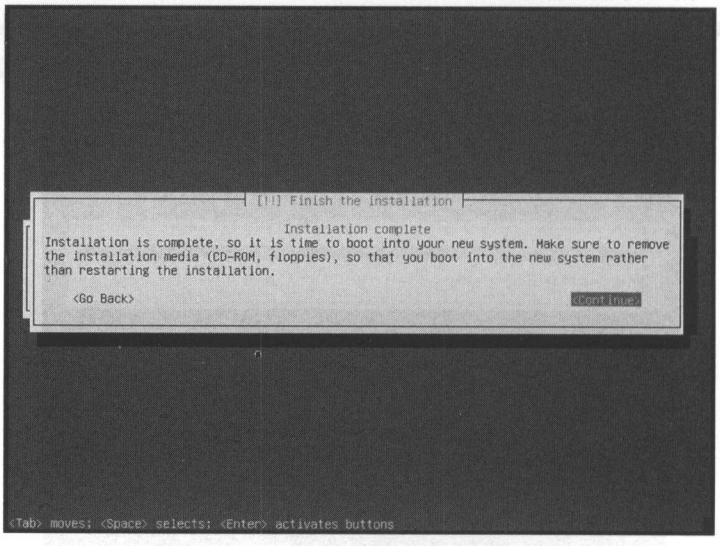


图 1-30

1.2　PXE 搭建（带 DHCP 模式）

请参照 1.1 节的安装方式，在 Ubuntu 系统安装完成后开始下面的配置。

（1）登录系统的配置，如图 1-31 所示。

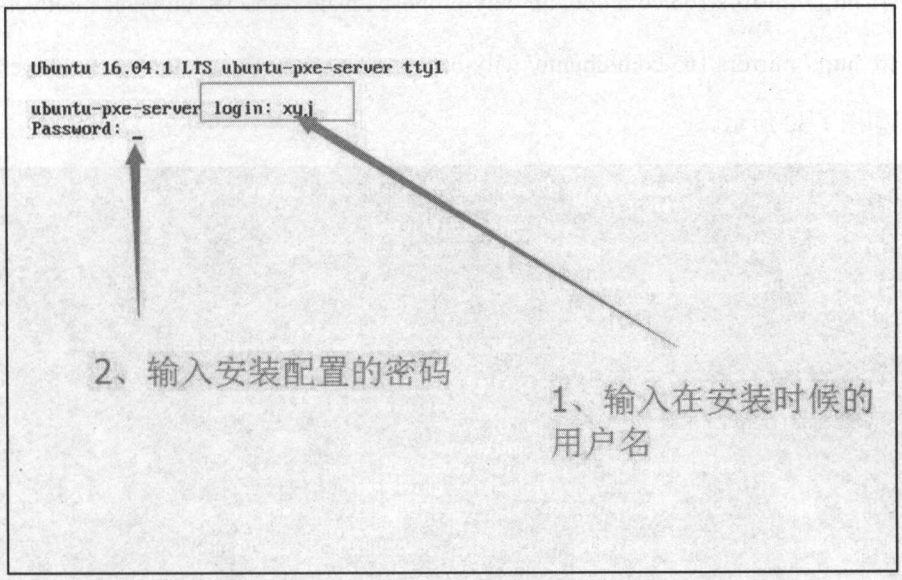


图 1-31

（2）切换国内 163 源。在使用过程中发现，之前安装时使用了香港的源，在不同网络中可能会有延迟，所以这里更换为国内 163 源。

```
$ sudo vim /etc/apt/sources.list #编辑源文件
```

把原来的内容都用“#”注释掉，新增下面的内容。

deb http://mirrors.163.com/ubuntu/ wily main restricted universe multiverse

deb http://mirrors.163.com/ubuntu/ wily-security main restricted universe multiverse

deb http://mirrors.163.com/ubuntu/ wily-updates main restricted universe multiverse

deb http://mirrors.163.com/ubuntu/ wily-proposed main restricted universe multiverse

deb http://mirrors.163.com/ubuntu/ wily-backports main restricted universe multiverse

deb-src http://mirrors.163.com/ubuntu/ wily main restricted universe multiverse

deb-src http://mirrors.163.com/ubuntu/ wily-security main restricted universe multiverse

deb-src http://mirrors.163.com/ubuntu/ wily-updates main restricted universe multiverse

deb-src http://mirrors.163.com/ubuntu/ wily-proposed main restricted universe multiverse

deb-src http://mirrors.163.com/ubuntu/ wily-backports main restricted universe multiverse

过程如图 1-32 所示。

```
# See http://help.ubuntu.com/community/UpgradeNotes for how to upgrade to
# newer versions of the distribution.
deb http://mirrors.163.com/ubuntu/ xenial main restricted
# deb-src http://hk.archive.ubuntu.com/ubuntu/ xenial main restricted

## Major bug fix updates produced after the final release of the
## distribution.
deb http://mirrors.163.com/ubuntu/ xenial-updates main restricted
# deb-src http://hk.archive.ubuntu.com/ubuntu/ xenial-updates main restricted

## N.B. software from this repository is ENTIRELY UNSUPPORTED by the Ubuntu
## team. Also, please note that software in universe WILL NOT receive any
## review or updates from the Ubuntu security team.
deb http://mirrors.163.com/ubuntu/ xenial universe
# deb-src http://hk.archive.ubuntu.com/ubuntu/ xenial universe
deb http://mirrors.163.com/ubuntu/ xenial-updates universe
# deb-src http://hk.archive.ubuntu.com/ubuntu/ xenial-updates universe

## N.B. software from this repository is ENTIRELY UNSUPPORTED by the Ubuntu
## team, and may not be under a free licence. Please satisfy yourself as to
## your rights to use the software. Also, please note that software in
## multiverse WILL NOT receive any review or updates from the Ubuntu
## security team.
deb http://mirrors.163.com/ubuntu/ xenial multiverse
# deb-src http://hk.archive.ubuntu.com/ubuntu/ xenial multiverse
deb http://mirrors.163.com/ubuntu/ xenial-updates multiverse
# deb-src http://hk.archive.ubuntu.com/ubuntu/ xenial-updates multiverse

## N.B. software from this repository may not have been tested as
## extensively as that contained in the main release, although it includes
## newer versions of some applications which may provide useful features.
## Also, please note that software in backports WILL NOT receive any review
## or updates from the Ubuntu security team.
deb http://mirrors.163.com/ubuntu/ xenial-backports main restricted universe multiverse
# deb-src http://hk.archive.ubuntu.com/ubuntu/ xenial-backports main restricted universe multiverse
                                                                    40,1          26%
```

图 1-32

（3）更新源和系统更新（一定要先更新源，才能找到新的补丁或软件），如图 1-33 所示。

```
$sudo apt-get update #更新源文件索引列表
$sudo apt-get upgrade #更新软件包
```

```
xyj@ubuntu-pxe-server:~$ sudo apt-get update
Hit:1 http://mirrors.163.com/ubuntu xenial InRelease
Hit:2 http://mirrors.163.com/ubuntu xenial-updates InRelease
Hit:3 http://mirrors.163.com/ubuntu xenial-backports InRelease
Get:4 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Fetched 102 kB in 1s (75.0 kB/s)
Reading package lists... Done
xyj@ubuntu-pxe-server:~$ sudo apt-get upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
The following packages have been kept back:
  snapd ubuntu-core-launcher
0 upgraded, 0 newly installed, 0 to remove and 2 not upgraded.
xyj@ubuntu-pxe-server:~$ _
```

图 1-33

（4）安装前置软件包 vim、inetutils-inetd 和 tftpd-hpa，如图 1-34 所示。

```
$sudo apt-get install -y vim apache2 inetutils-inetd tftpd-hpa
```

```
xyj@ubuntu-pxe-server:~$ sudo apt-get install -y vim apache2 inetutils-inetd tftpd-hpa _
```

图 1-34

（5）安装 DHCP 服务，如图 1-35 所示。

```
$sudo apt-get install -y isc-dhcp-server
```

```
xyj@ubuntu-pxe-server:~$ sudo apt-get install -y is
isag                    iscsitarget-dkms        isenkram-cli            ispanish
isakmpd                 isdnactivecards         isight-firmware-tools   ispell
isatapd                 isdnlog                 islamic-menus           isrcsubmit
isc-dhcp-client         isdnlog-data            ismrmrd-schema          isso
isc-dhcp-client-ddns    isdnutils-base          ismrmrd-tools           istgt
isc-dhcp-common         isdnutils-doc           iso-codes               iswedish
isc-dhcp-dbg            isdnutils-xtools        isolinux                iswiss
isc-dhcp-dev            isdnvbox                isomaster               isympy
isc-dhcp-relay          isdnvboxclient          isomd5sum               isync
isc-dhcp-server         isdnvboxserver          isoqlog
isc-dhcp-server-ldap    iselect                 isoquery
iscsitarget             isenkram                isort
xyj@ubuntu-pxe-server:~$ sudo apt-get install -y isc-dhcp-server
```

图 1-35

（6）tftpd-hpa 是一个功能增强的 TFTP 服务器。配置 tftpd-hpa，新增两行使 TFTP 服务在后台运行（默认不运行），如图 1-36 所示。

```
$sudo vim /etc/default/tftpd-hpa
```

```
xyj@ubuntu-pxe-server:~$ sudo vim /etc/default/tftpd-hpa
```

图 1-36

```
TFTP_ADDRESS="192.168.2.254:69" #固定为服务器 IP

RUN_DAEMON="yes"  #新增内容
OPTIONS="-l -s /var/lib/tftpboot"  #新增内容
```

TFTP 配置文件修改如图 1-37 所示。

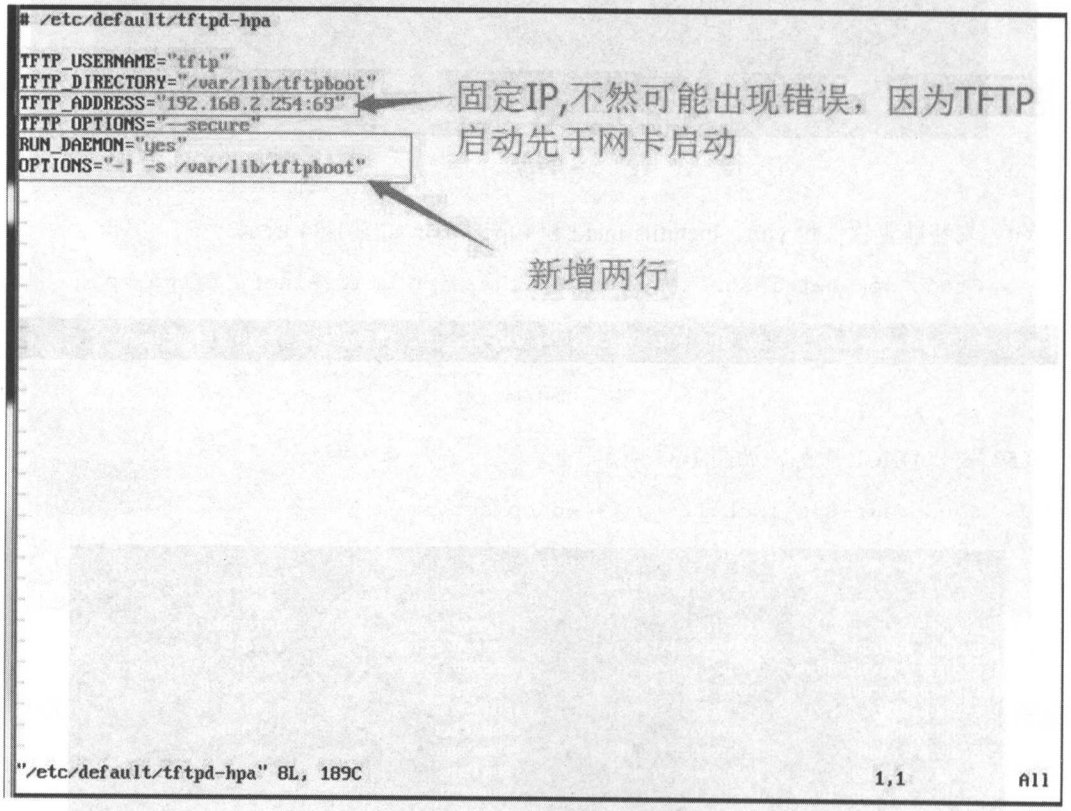


图 1-37

保存后重启守护进程，如图 1-38 所示。

```
$sudo /etc/init.d/tftpd-hpa restart #重启 TFTP 服务
```

```
xyj@ubuntu-pxe-server:~$ sudo /etc/init.d/tftpd-hpa restart
[sudo] password for xyj:
[ ok ] Restarting tftpd-hpa (via systemctl): tftpd-hpa.service.
xyj@ubuntu-pxe-server:~$
```

图 1-38

查看 TFTP 服务状态，如图 1-39 所示。

```
xyj@ubuntu-pxe-server:~$ sudo systemctl status tftpd-hpa.service
● tftpd-hpa.service - LSB: HPA's tftp server
   Loaded: loaded (/etc/init.d/tftpd-hpa; bad; vendor preset: enabled)
   Active: active (running) since Sat 2016-12-24 17:19:42 CST; 31min ago
     Docs: man:systemd-sysv-generator(8)
  Process: 3288 ExecStop=/etc/init.d/tftpd-hpa stop (code=exited, status=0/SUCCESS)
  Process: 3299 ExecStart=/etc/init.d/tftpd-hpa start (code=exited, status=0/SUCCESS)
    Tasks: 1
   Memory: 156.0K
      CPU: 31ms
   CGroup: /system.slice/tftpd-hpa.service
           └─3309 /usr/sbin/in.tftpd --listen --user tftp --address [::]:69 --secure /var/lib/tftpbo

Dec 24 17:19:42 ubuntu-pxe-server systemd[1]: Starting LSB: HPA's tftp server...
Dec 24 17:19:42 ubuntu-pxe-server tftpd-hpa[3299]:  * Starting HPA's tftpd in.tftpd
Dec 24 17:19:42 ubuntu-pxe-server tftpd-hpa[3299]:    ...done.
Dec 24 17:19:42 ubuntu-pxe-server systemd[1]: Started LSB: HPA's tftp server.
lines 1-16/16 (END)
```

图 1-39

（7）对外开启 TFTP 服务（默认对外端口为禁止），编辑/etc/inetd.conf 文件，新增一行，如图 1-40 所示。

```
$sudo vim /etc/inetd.conf
```

```
xyj@ubuntu-pxe-server:~$ sudo vim /etc/inetd.conf
```

图 1-40

新增 TFTP 监视代码，如图 1-41 所示。

```
tftp    dgram   udp     wait    root    /usr/sbin/in.tftpd
/usr/sbin/in.tftpd -s /var/lib/tftpboot
```

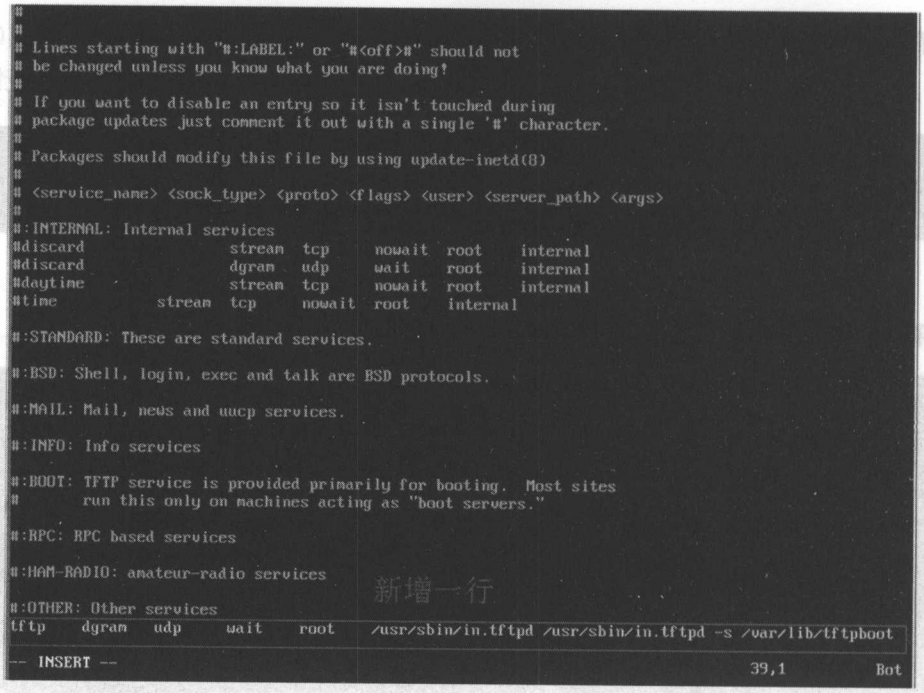


图 1-41

（8）挂载光驱并查看光驱内容，如图 1-42 所示。默认不挂载，所以找不到光盘。

```
$sudo mount /dev/cdrom /media/cdrom/ #将光驱挂载到/media/cdrom/目录下
$ls /media/cdrom/ #查看挂载光驱内容
```

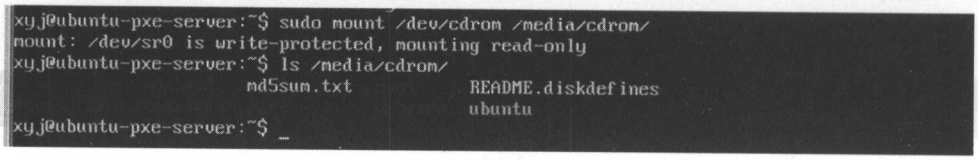


图 1-42

（9）复制 Ubuntu 16.04 Server 镜像引导文件到 TFTP 目录下，如图 1-43 所示。

```
$sudo cp -fr /media/cdrom/install/netboot/* /var/lib/tftpboot/ #将光驱内网络引导所有内容复制到 tftp 目录下
```

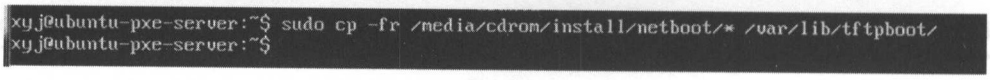


图 1-43

（10）复制 Ubuntu 16.04 Server 镜像，如图 1-44 所示。

```
$sudo mkdir /var/www/html/ubuntu #新建 Ubuntu 镜像目录
$sudo cp -fr /media/cdrom/* /var/www/html/ubuntu/ #复制 Ubuntu 镜像
```

```
xyj@ubuntu-pxe-server:~$ sudo mkdir /var/www/html/ubuntu
xyj@ubuntu-pxe-server:~$ sudo cp -fr /media/cdrom/* /var/www/html/ubuntu/
xyj@ubuntu-pxe-server:~$ _
```

图 1-44

（11）配置静态 IP，测试服务器为双网卡，现在将其中一张（ens35）改为静态 IP，如图 1-45 所示。

```
$ifconfig #查看网络配置信息
$sudo vim /etc/network/interfaces #编辑网络配置
```

```
xyj@ubuntu-pxe-server:~$ ifconfig
ens32     Link encap:Ethernet  HWaddr 00:0c:29:66:6f:15
          inet addr:192.168.31.192  Bcast:192.168.31.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe66:6f15/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:241 errors:0 dropped:0 overruns:0 frame:0
          TX packets:26 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:19464 (19.4 KB)  TX bytes:2530 (2.5 KB)

ens35     Link encap:Ethernet  HWaddr 00:0c:29:66:6f:1f
          inet addr:192.168.182.131  Bcast:192.168.182.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe66:6f1f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:24 errors:0 dropped:0 overruns:0 frame:0
          TX packets:14 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2860 (2.8 KB)  TX bytes:1856 (1.8 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:161 errors:0 dropped:0 overruns:0 frame:0
          TX packets:161 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:11889 (11.8 KB)  TX bytes:11889 (11.8 KB)

xyj@ubuntu-pxe-server:~$ sudo vim /etc/network/interfaces
```

图 1-45

将 ens35 网卡设置为固定 IP，如图 1-46 所示。

```
auto ens35
#iface ens35 inet dhcp  #注释动态获取 IP
iface ens35 inet static #更改为静态 IP
address 192.168.2.254 #设置 IP 地址
netmask 255.255.255.0 #子网掩码
gateway 192.168.2.1 #网关
```

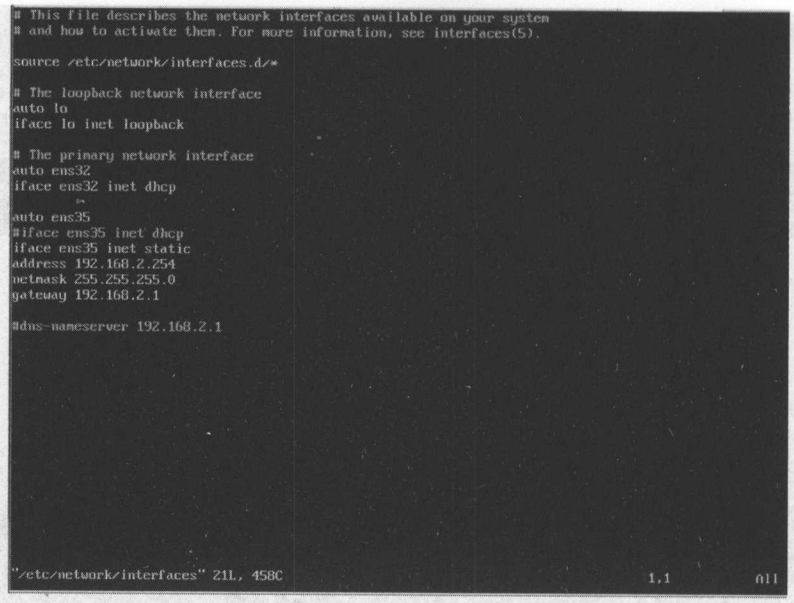


图 1-46

（12）修改网卡 ens35，使其提供 DHCP 服务，如图 1-47 所示。

```
$sudo /etc/default/isc-dhcp-server #设置 DHCP 服务工作网卡
```

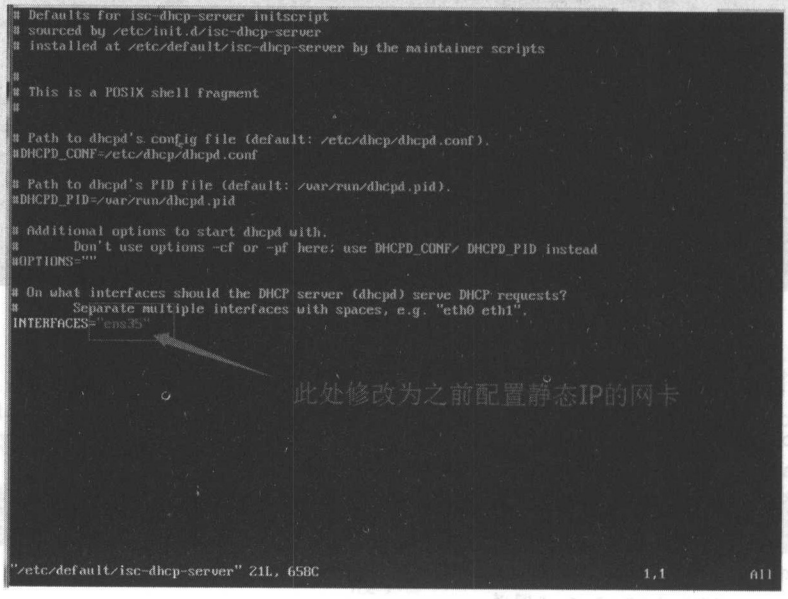


图 1-47

（13）配置 DHCP 地址池，如图 1-48 所示。

```
$sudo /etc/dhcp/dhcpd.conf #DHCP 详细配置
subnet 192.168.2.0 netmask 255.2.55.255.0{
  range 192.168.2.100 192.168.2.200; #分配 IP 段范围
  option subnet-mask 255.255.255.0; #子网掩码
  option routers 192.168.2.254; #默认网关
  option broadcast-address 192.168.2.255;#广播地址
}
```

```
#
# Sample configuration file for ISC dhcpd for Debian
#
# Attention: If /etc/ltsp/dhcpd.conf exists, that will be used as
# configuration file instead of this file.
#
#


# The ddns-updates-style parameter controls whether or not the server will
# attempt to do a DNS update when a lease is confirmed. We default to the
# behavior of the version 2 packages ('none', since DHCP v2 didn't
# have support for DDNS.)
ddns-update-style none;

# option definitions common to all supported networks...
option domain-name "example.org";
option domain-name-servers ns1.example.org, ns2.example.org;

default-lease-time 600;
max-lease-time 7200;

subnet 192.168.2.0 netmask 255.255.255.0 {
        range 192.168.2.100 192.168.2.200;
        option subnet-mask 255.255.255.0;
        option routers 192.168.2.254;
        option broadcast-address 192.168.2.255;
        #option domain-name-servers 192.168.2.1;
}

# If this DHCP server is the official DHCP server for the local
# network, the authoritative directive should be uncommented.
#authoritative;

# Use this to send dhcp log messages to a different log file (you also
# have to hack syslog.conf to complete the redirection).
log-facility local7;
"/etc/dhcp/dhcpd.conf" 119L, 3872C                                    1,1           Top
```

分配子网

分配IP范围

子网掩码。

广播地址

图 1-48

（14）在 DHCP 中配置 PXE 引导。

```
$sudo /etc/dhcp/dhcpd.conf
next-server 192.168.2.254;
filename "pxelinux.0";
```

新增位置如图 1-49 所示。

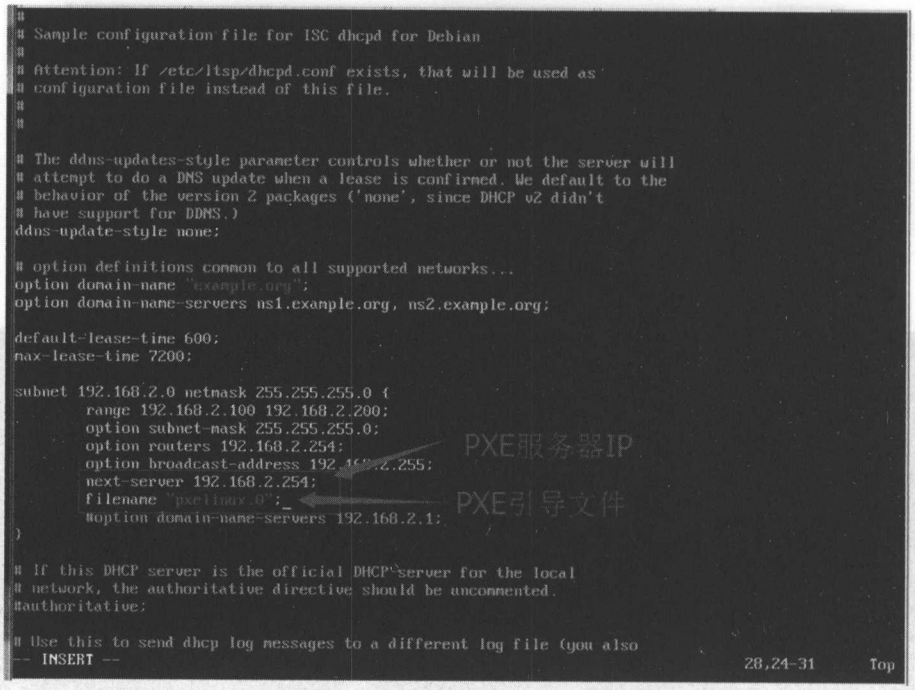


图 1-49

（15）新建虚拟机，使用网络启动，如图 1-50 所示。

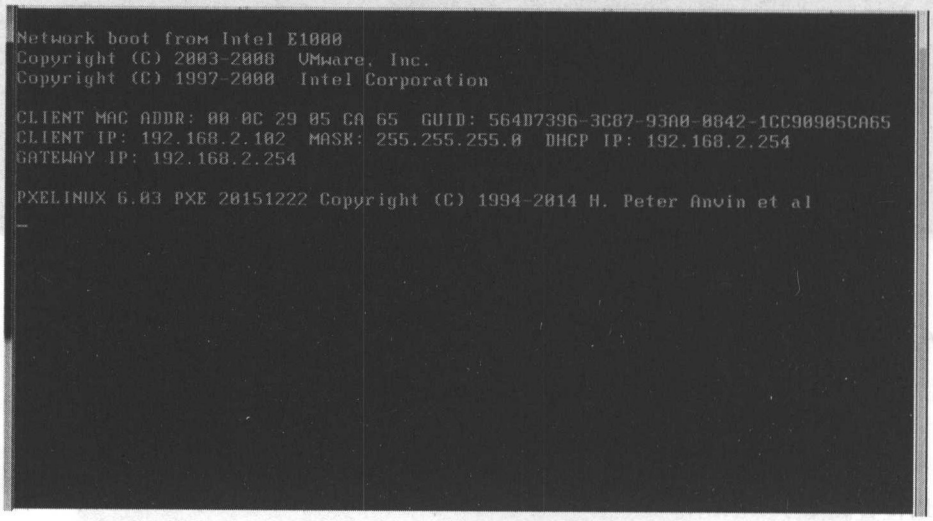


图 1-50

PXE 引导选择安装 Ubuntu 系统，如图 1-51 所示。

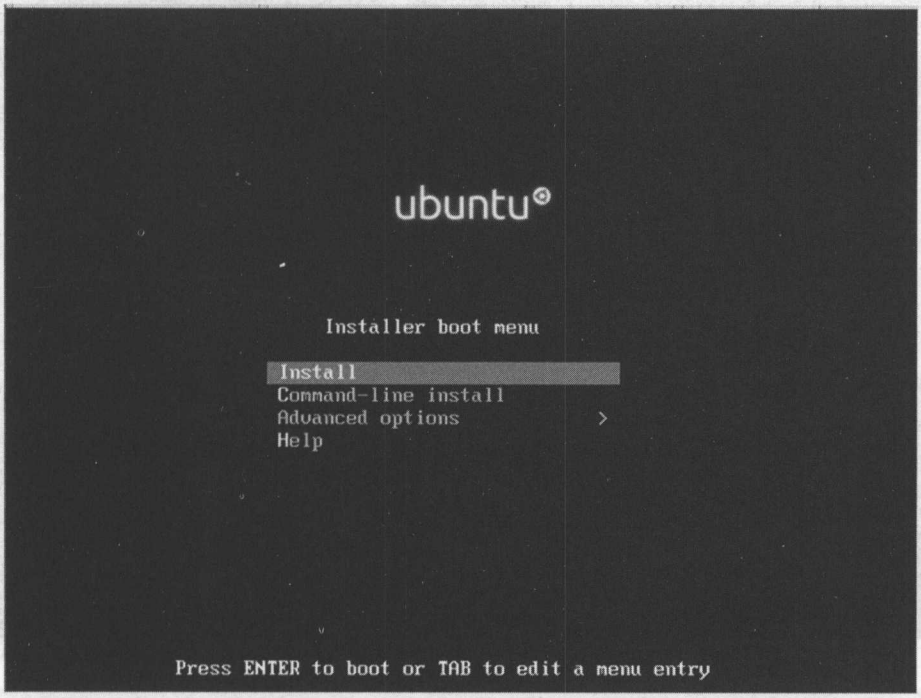


图 1-51

至此，带 DHCP 的 PXE 服务器搭建完成。

1.3　PXE 搭建（DHCP 不可控）

有时候我们所在的网络环境是通过路由器自动获取 IP 的，这时如果按照 1.2 节介绍的方法搭建 PXE，就会出现 DHCP 冲突的问题，所以这里使用了官方推荐使用的 ProxyDHCP 解决方案。

（1）删除 dhcp3-server 包，卸载 DHCP 服务，如图 1-52 所示（默认是未安装，这继续使用前面的虚拟机配置，新服务器可以忽略这一步）。

```
$sudo apt-get -y --auto-remove purge isc-dhcp-server #卸载 DHCP 服务
```

```
xyj@ubuntu-pxe-server:~$ sudo apt-get -y --auto-remove purge isc-dhcp-server
[sudo] password for xyj:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be REMOVED:
  isc-dhcp-server* libirs-export141* libisccfg-export140*
0 upgraded, 0 newly installed, 3 to remove and 2 not upgraded.
After this operation, 1,583 kB disk space will be freed.
```

图 1-52

（2）安装 dnsmasq 包，dnsmasq 是一个“简单”的 Linux 工具，整合了 DNS 服务器，可选择 DHCP 功能。

```
$sudo apt-get install -y dnsmasq
```

```
xyj@ubuntu-pxe-server:~$ sudo apt-get install -y dnsmasq
[sudo] password for xyj:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  dnsmasq
0 upgraded, 1 newly installed, 0 to remove and 2 not upgraded.
Need to get 15.9 kB of archives.
After this operation, 71.7 kB of additional disk space will be used.
Get:1 http://mirrors.163.com/ubuntu xenial-updates/universe amd64 dnsmasq all 2.75-1ubuntu0.16.04.1
[15.9 kB]
```

图 1-53

提示：安装好 dnsmasq 后将失去 DNS 获取功能，如果出现无法访问网络的情况，则关闭 dnsmasq，重新获取 DNS 即可访问网络。

（3）配置 dnsmasq。

```
$sudo vim /etc/dnsmasq.conf
port=0
dhcp-range=192.168.1.0,proxy
dhcp-boot=pxelinux.0
tftp-root=/var/lib/tftpboot
log-dhcp
```

禁用 DNS 功能，如图 1-54 所示。

```
# Listen on this specific port instead of the standard DNS port
# (53). Setting this to zero completely disables DNS function,
# leaving only DHCP and/or TFTP.
port 0
# The following two options make you a better netizen, since they
# tell dnsmasq to filter out queries which the public DNS cannot
# answer, and which load the servers (especially the root servers)
# unnecessarily. If you have a dial-on-demand link they also stop
# these requests from bringing up the link unnecessarily.

# Never forward plain names (without a dot or domain part)
#domain-needed
# Never forward addresses in the non-routed address spaces.
#bogus-priv

# Uncomment these to enable DNSSEC validation and caching:
# (Requires dnsmasq to be built with DNSSEC option.)
#conf-file=%%PREFIX%%/share/dnsmasq/trust-anchors.conf
#dnssec

# Replies which are not DNSSEC signed may be legitimate, because the domain
# is unsigned, or may be forgeries. Setting this option tells dnsmasq to
# check that an unsigned reply is OK, by finding a secure proof that a DS
# record somewhere between the root and the domain does not exist.
# The cost of setting this is that even queries in unsigned domains will need
# one or more extra DNS queries to verify.
#dnssec-check-unsigned

# Uncomment this to filter useless windows-originated DNS requests
-- INSERT --                                                    10,7          Top
```

将监听端口设置为0，禁用DNS功能

图 1-54

设置 IP 生效范围，如图 1-55 所示。

```
# Uncomment this to enable the integrated DHCP server, you need
# to supply the range of addresses available for lease and optionally
# a lease time. If you have more than one network, you will need to
# repeat this for each network on which you want to supply DHCP
# service.
#dhcp-range=192.168.0.50,192.168.0.150,12h
dhcp-range 192.168.31.0 proxy
# This is an example of a DHCP range where the netmask is given. This
# is needed for networks we reach the dnsmasq DHCP server via a relay
# agent. If you don't know what a DHCP relay agent is, you probably
# don't need to worry about this.
#dhcp-range=192.168.0.50,192.168.0.150,255.255.255.0,12h

# This is an example of a DHCP range which sets a tag, so that
# some DHCP options may be set only for this network.
#dhcp-range=set:red,192.168.0.50,192.168.0.150

# Use this DHCP range only when the tag "green" is set.
#dhcp-range=tag:green,192.168.0.50,192.168.0.150,12h

# Specify a subnet which can't be used for dynamic address allocation,
# is available for hosts with matching --dhcp-host lines. Note that
# dhcp-host declarations will be ignored unless there is a dhcp-range
-- INSERT --                                                    159,1         22%
```

新增一行或把上面一行反注释后修改为当前网络的IP段

图 1-55

设置 PEX 引导程序，如图 1-56 所示。

```
# Set the boot filename for netboot/PXE. You will only need
# this is you want to boot machines over the network and you will need
# a TFTP server; either dnsmasq's built in TFTP server or an
# external one. (See below for how to enable the TFTP server.)
dhcp-boot=pxelinux.0

# The same as above, but use custom tftp-server instead machine running dnsmasq
#dhcp-boot=pxelinux,server.name,192.168.1.100

# Boot for Etherboot gPXE. The idea is to send two different
# filenames, the first loads gPXE, and the second tells gPXE what to
# load. The dhcp-match sets the gpxe tag for requests from gPXE.
#dhcp-match=set:gpxe,175 # gPXE sends a 175 option.
#dhcp-boot=tag:!gpxe,undionly.kpxe
#dhcp-boot=mybootimage

# Encapsulated options for Etherboot gPXE. All the options are
# encapsulated within option 175
#dhcp-option=encap:175, 1, 5b         # priority code
#dhcp-option=encap:175, 176, 1b       # no-proxydhcp
#dhcp-option=encap:175, 177, string   # bus-id
#dhcp-option=encap:175, 189, 1b       # BIOS drive code
#dhcp-option=encap:175, 190, user     # iSCSI username
-- INSERT --                                                    443,1         67%
```

图 1-56

设置 TFTP 根目录位置，如图 1-57 所示。

```
# Do real PXE, rather than just booting a single file, this is an
# alternative to dhcp-boot.
#pxe-prompt="What system shall I netboot?"
# or with timeout before first available action is taken:
#pxe-prompt="Press F8 for menu.", 60

# Available boot services. for PXE.
#pxe-service=x86PC, "Boot from local disk"

# Loads <tftp-root>/pxelinux.0 from dnsmasq TFTP server.
pxe-service=x86PC, "Install Linux", pxelinux

# Loads <tftp-root>/pxelinux.0 from TFTP server at 1.2.3.4.
# Beware this fails on old PXE ROMS.
#pxe-service=x86PC, "Install Linux", pxelinux, 1.2.3.4

# Use bootserver on network, found my multicast or broadcast.
#pxe-service=x86PC, "Install windows from RIS server", 1

# Use bootserver at a known IP address.
#pxe-service=x86PC, "Install windows from RIS server", 1, 1.2.3.4

# If you have multicast-FTP available,
# information for that can be passed in a similar way using options 1
# to 5. See page 19 of
# http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf


# Enable dnsmasq's built-in TFTP server
enable-tftp

# Set the root directory for files available via FTP.
tftp-root=/var/lib/tftpboot

# Do not abort if the tftp-root is unavailable
#tftp-no-fail
-- INSERT --                                                    481,1         74%
```

修改为tftpboot的目录，这里配置的是/var/lib/tftpboot

图 1-57

开启 DHCP 记录，如图 1-58 所示。

```
# For debugging purposes, log each DNS query as it passes through
# dnsmasq.
#log-queries

# Log lots of extra information about DHCP transactions.
log-dhcp

# Include another lot of configuration options.
#conf-file=/etc/dnsmasq.more.conf
#conf-dir=/etc/dnsmasq.d

# Include all the files in a directory except those ending in .bak
#conf-dir=/etc/dnsmasq.d,.bak

# Include all files in a directory which end in .conf
#conf-dir=/etc/dnsmasq.d/,*.conf
-- INSERT --                                              656,1         Bot
```

开启DHCP记录

图 1-58

（4）编辑/etc/default/dnsmasq，在结尾新增一行。

```
$sudo vim /etc/default/dnsmasq
```

新增如下内容，解决 nameserver 自动变 127.0.0.1 的问题，如图 1-59 所示。

```
DNSMASQ_EXCEPT=lo
```

```
# This file has five functions:
# 1) to completely disable starting dnsmasq,
# 2) to set DOMAIN_SUFFIX by running `dnsdomainname`
# 3) to select an alternative config file
#    by setting DNSMASQ_OPTS to --conf-file=<file>
# 4) to tell dnsmasq to read the files in /etc/dnsmasq.d for
#    more configuration variables.
# 5) to stop the resolvconf package from controlling dnsmasq's
#    idea of which upstream nameservers to use.
# For upgraders from very old versions, all the shell variables set
# here in previous versions are still honored by the init script
# so if you just keep your old version of this file nothing will break.

#DOMAIN_SUFFIX=`dnsdomainname`
#DNSMASQ_OPTS="--conf-file=/etc/dnsmasq.alt"

# Whether or not to run the dnsmasq daemon; set to 0 to disable.
ENABLED=1

# By default search this drop directory for configuration options.
# Libvirt leaves a file here to make the system dnsmasq play nice.
# Comment out this line if you don't want this. The dpkg-* are file
# endings which cause dnsmasq to skip that file. This avoids pulling
# in backups made by dpkg.
CONFIG_DIR=/etc/dnsmasq.d,.dpkg-dist,.dpkg-old,.dpkg-new

# If the resolvconf package is installed, dnsmasq will use its output
# rather than the contents of /etc/resolv.conf to find upstream
# nameservers. Uncommenting this line inhibits this behaviour.
# Note that including a "resolv-file=<filename>" line in
# /etc/dnsmasq.conf is not enough to override resolvconf if it is
# installed: the line below must be uncommented.
#IGNORE_RESOLVCONF=yes
DNSMASQ_EXCEPT=lo
                                                        34,17         All
```

新增一行

图 1-59

（5）重启 dnsmasq 服务，如图 1-60 所示。

```
$sudo service dnsmasq restart #重启 dnsmasq 服务
```

```
xyj@ubuntu-pxe-server:~$ sudo service dnsmasq restart
xyj@ubuntu-pxe-server:~$ _
```

图 1-60

（6）新建虚拟机，使用网络启动，如图 1-61 所示。

```
Network boot from Intel E1000
Copyright (C) 2003-2008  VMware, Inc.
Copyright (C) 1997-2000  Intel Corporation

CLIENT MAC ADDR: 00 0C 29 05 CA 65  GUID: 564D7396-3C87-93A0-0842-1CC90905CA65
CLIENT IP: 192.168.31.183  MASK: 255.255.255.0
DHCP IP: 192.168.31.1  PROXY IP: 192.168.31.192
GATEWAY IP: 192.168.31.1

Auto-select:
    Install Linux

BOOT SERVER IP: 192.168.31.192

PXELINUX 6.03 PXE 20151222 Copyright (C) 1994-2014 H. Peter Anvin et al
_
```

图 1-61

1.4 KickStart 无人职守配置

前面安装好了 PXE 服务器，但这只是提供了远程镜像，还需要人工去服务器上面安装配置。有什么方法可以自动完成呢？该 KickStart 出场了。

KickStart 通过 ks.cfg 文件预先把系统的参数都配置好，网络引导开始后就会去查找 ks.cfg 文件，如果有这个文件则自动安装系统，没有就需要人工干预安装系统。

（1）安装 system-config-kickstart 工具包，可以图形化配置 KickStart，如图 1-62 所示。

```
$sudo apt-get update
$sudo apt-get upgrade
```

```
$sudo apt-get install -y system-config-kickstart
```

```
xyj@xyj-virtual-machine: ~
xyj@xyj-virtual-machine:~$ sudo apt-get install -y system-config-kickstart
[sudo] xyj 的密码：
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会同时安装下列软件：
  isoquery libisocodes1 localechooser-data python-apt
建议安装：
  python-apt-dbg python-apt-doc
下列【新】软件包将被安装：
  isoquery libisocodes1 localechooser-data python-apt system-config-kickstart
升级了 0 个软件包，新安装了 5 个软件包，要卸载 0 个软件包，有 12 个软件包未被升级。
有 2 个软件包没有被完全安装或卸载。
需要下载 570 kB 的归档。
```

图 1-62

（2）安装完毕后输入 system-config-kickstart 即可打开 KickStart，如图 1-63 所示。

```
$system-config-kickstart
```

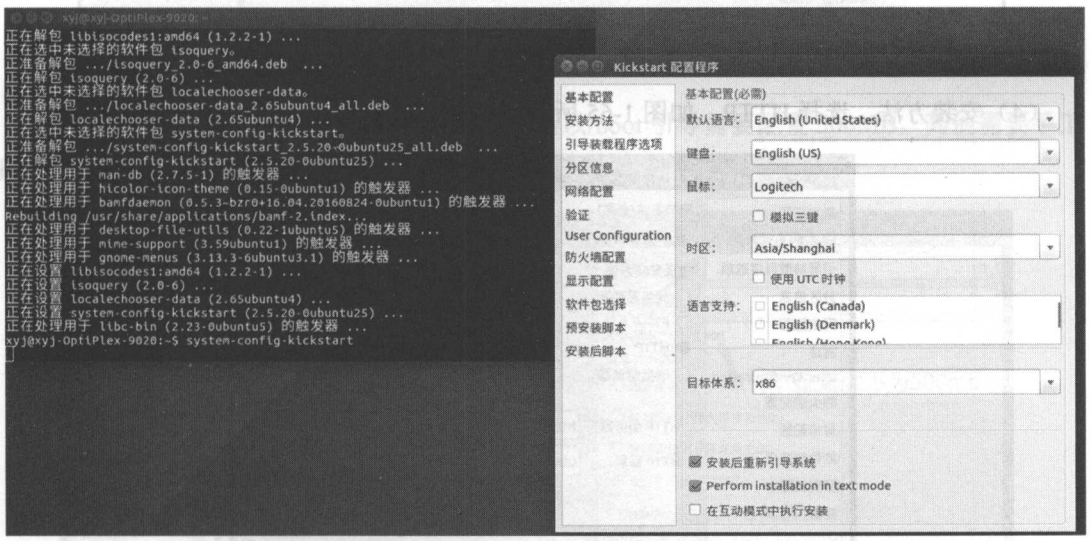


图 1-63

（3）基本配置如图 1-64 所示。

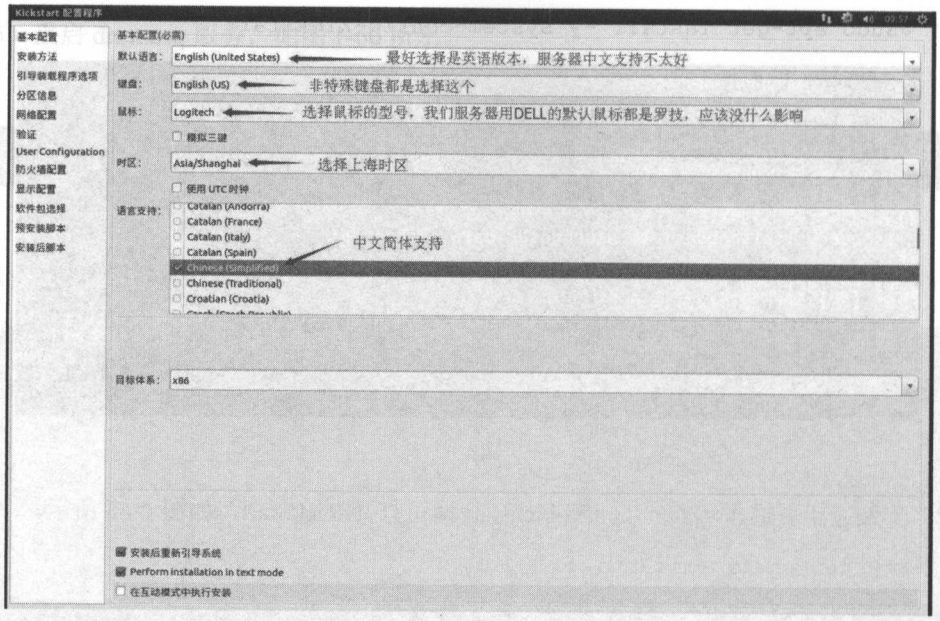


图 1-64

（4）安装方法，选择 HTTP，如图 1-65 所示。

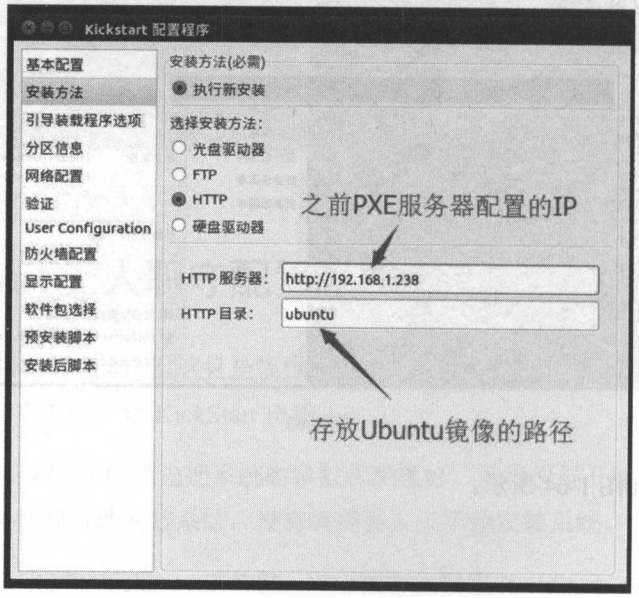


图 1-65

（5）引导方式默认即可，如图 1-66 所示。

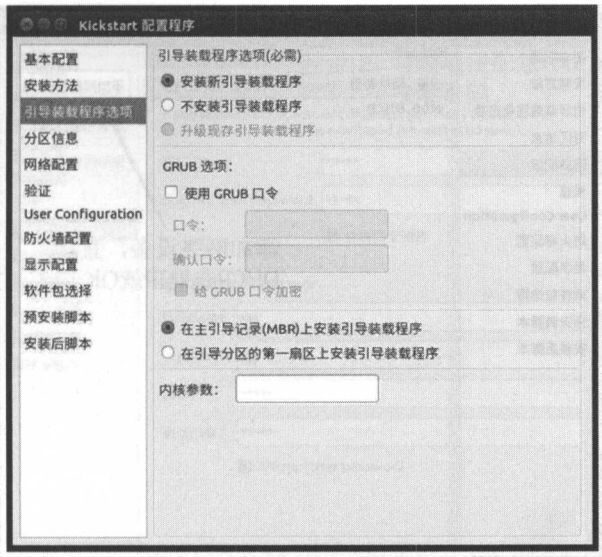


图 1-66

（6）分区信息配置，这里采用 LVM 配置，所以/boot 引导需要配置 500MB，其他需要手工配置，如图 1-67 所示。

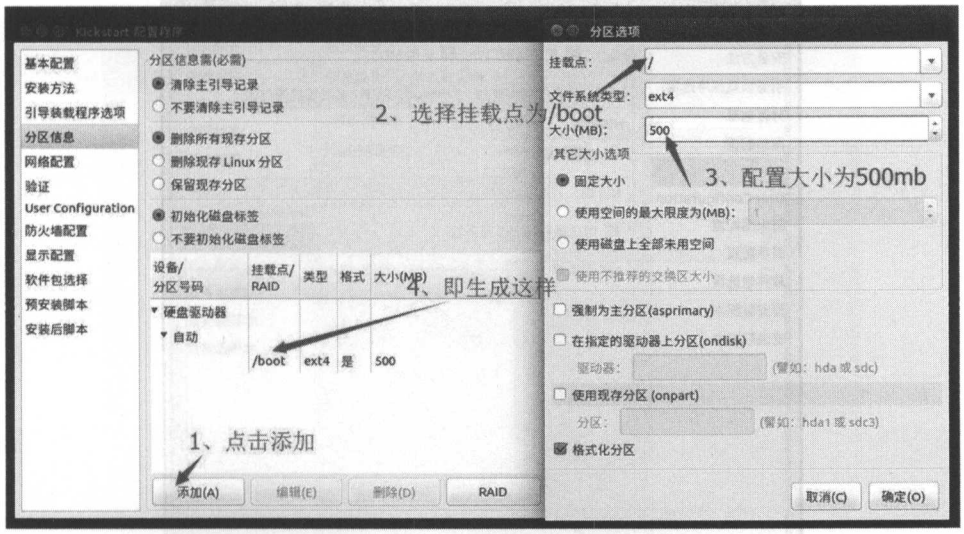


图 1-67

（7）网络配置如图 1-68 所示。

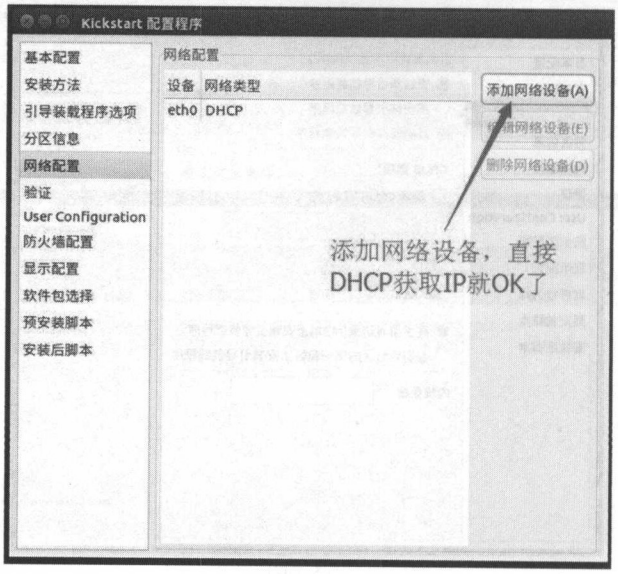


图 1-68

（8）除非使用了 NIS、LDAP“需要验证”才配置，一般直接略过，如图 1-69 所示。

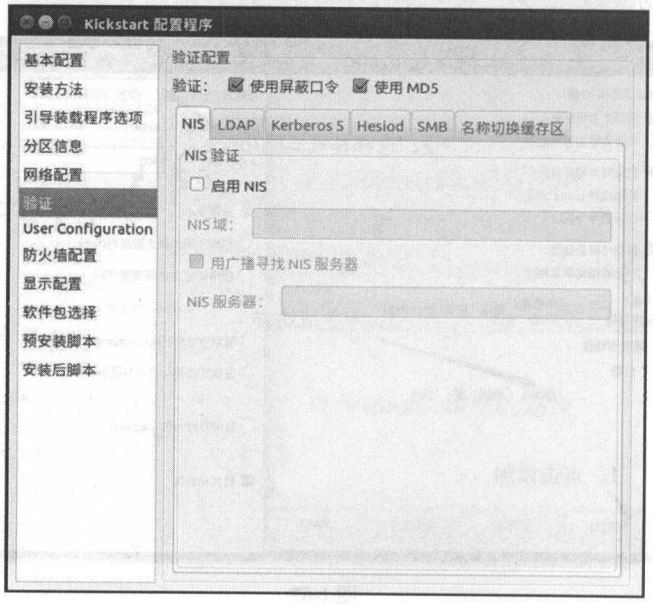


图 1-69

（9）配置账号信息，如图 1-70 所示。

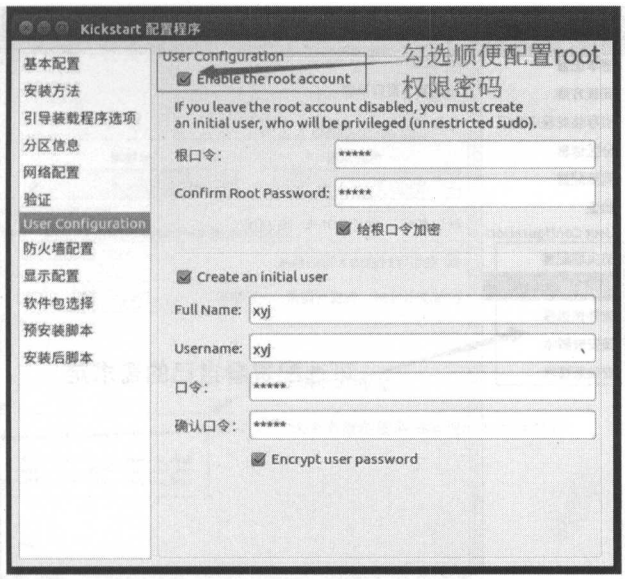


图 1-70

（10）防火墙默认是禁用的，除非有特殊需求才配置，如图 1-71 所示。

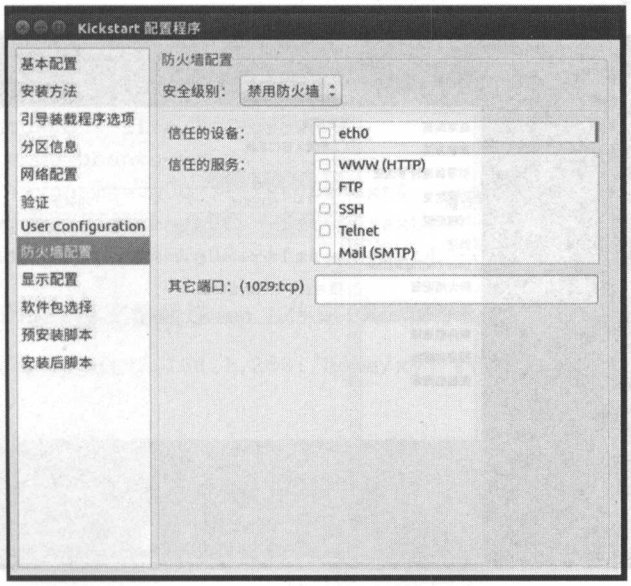


图 1-71

（11）可选配置，根据自己实际需求进行配置，如图 1-72 所示。

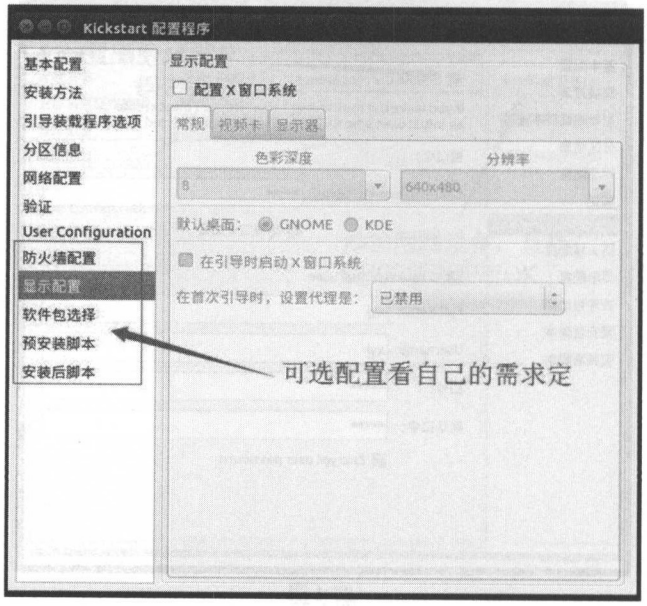


图 1-72

（12）保存 ks.cfg 文件，如图 1-73 所示。

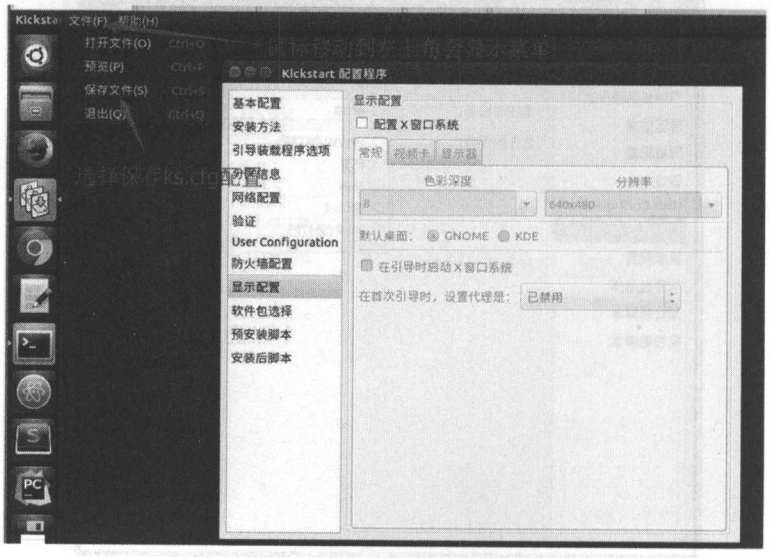


图 1-73

（13）找到刚刚保存的 ks.cfg 文件，手动修改 LVM 配置，如图 1-74 所示。

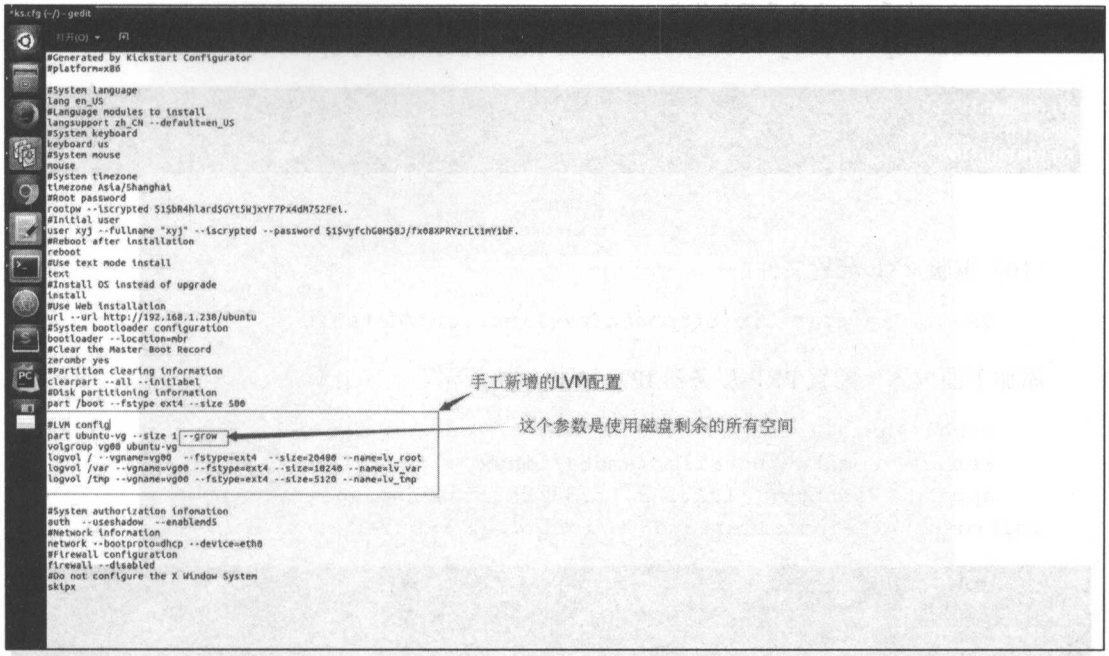


图 1-74

插入下列代码配置 LVM。

```
#LVM config
part ubuntu-vg --size 1 --grow
volgroup vg00 ubuntu-vg
logvol / --vgname=vg00  --fstype=ext4  --size=20480 --name=lv_root
logvol /var --vgname=vg00 --fstype=ext4 --size=10240 --name=lv_var
logvol /tmp --vgname=vg00 --fstype=ext4 --size=5120 --name=lv_tmp
```

（14）将配置好的 ks.cfg 文件通过 scp 上传到 PXE 服务器上，如图 1-75 所示。

```
$scp ks.cfg xyj@192.168.1.238:/home/xyj #将配置好的 ks 文件通过 scp 上传到
#PXE 服务器
```

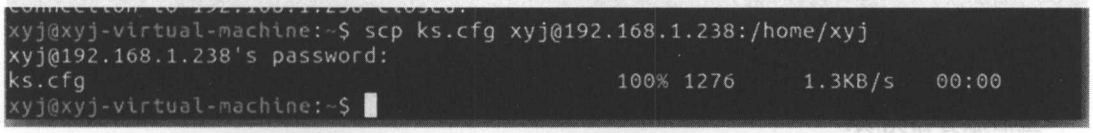


图 1-75

（15）在 PXE 服务器上操作，将 ks.cfg 文件放到/var/www/html/下，如图 1-76 所示。

```
$ls #查看 ks 文件是否上传成功
$sudo cp ks.cfg /var/www/html/ #将 ks 文件复制到 apache 服务目录下
```

```
xyj@ubuntu-pxe-server:~$ ls
ks.cfg
xyj@ubuntu-pxe-server:~$ _
```

图 1-76

（16）编辑 PXE 配置文件。

```
$sudo vim /var/lib/tftpboot/pxelinux.cfg/default
```

添加下面内容，配置 PXE 服务器 IP，如图 1-77 所示。

```
    label Install Ubuntu16.04_server
    kernel  ubuntu-installer/amd64/linux
    append  ks=http://192.168.1.238/ks.cfg vga=normal
initrd=ubuntu-installer/amd64 /initrd.gz --quiet
```

```
# D-I config version 2.0
# search path for the c32 support libraries (libcom32, libutil etc.)
path ubuntu-installer/amd64/boot-screens/
include ubuntu-installer/amd64/boot-screens/menu.cfg
default ubuntu-installer/amd64/boot-screens/vesamenu.c32
prompt 0
timeout 0
label Install Ubuntu16.04_server
       kernel  ubuntu-installer/amd64/linux
       append  ks=http://192.168.1.238/ks.cfg  vga=normal initrd=ubuntu-installer/amd64/initrd.gz --quiet
~
~
~
```

1.5　使用 PXE 安装系统

（1）在 BIOS 中修改启动方式为网络启动，如图 1-78 所示。

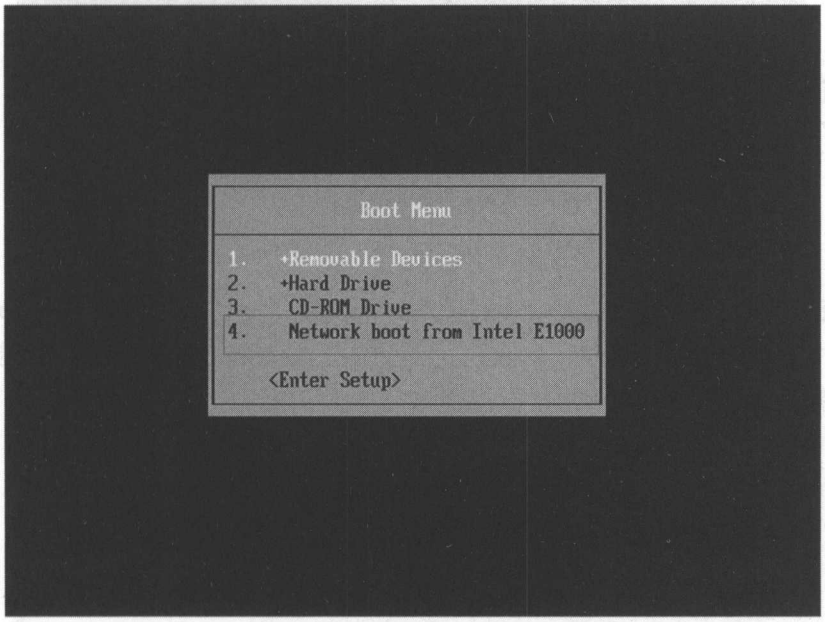


图 1-78

（2）寻找 PXE 服务，如图 1-79 所示。

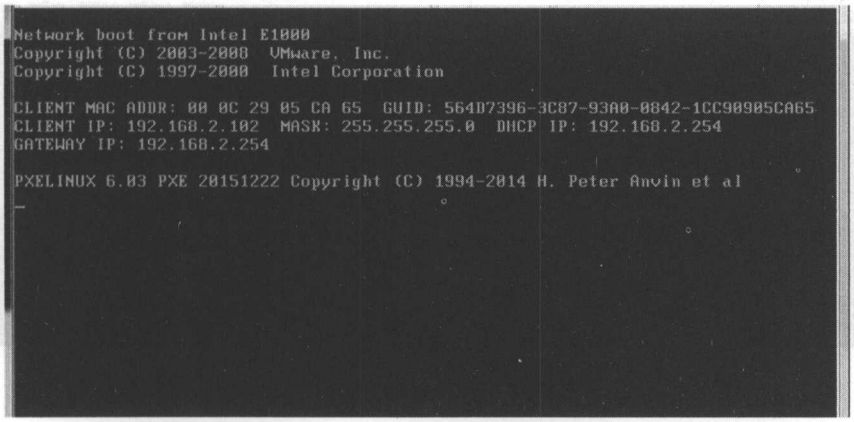


图 1-79

（3）选择自动安装 Ubuntu，如图 1-80 所示，单击“Install Ubuntu 16.04_server”，其他的交给系统自动完成。

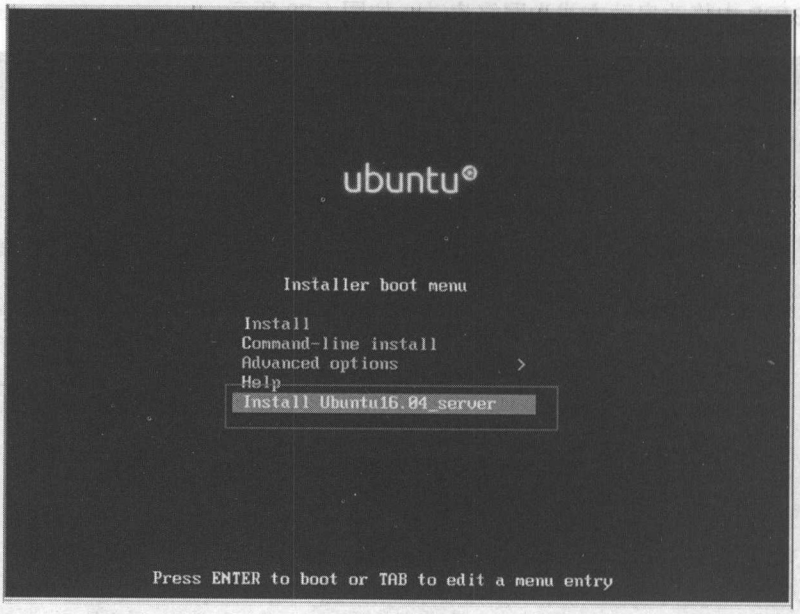


图 1-80

第 2 章

Python开发工具——sublime3使用

sublime3 是轻量级文本编辑器，可以跨平台启动且速度很快，因此可称之为轻兵器。虽然 sublime3 能够运行 Python 解释器，但是不推荐在 sublime3 上运行 Python 脚本。为什么选择 sublime3 而不选择 VSCode 呢？如图 2-1 至图 2-3 所示（里面测试数据仅为个人电脑测试）。

软件	费用	安装包	启动速度（个人电脑测试，不能作为任何测试依据）	打开大文件
Sublime Text 3126	收费	8.07M	快（1.8秒）	能加载
VSCode 1.8.1	免费开源	32.43M	较快（4秒）	不能加载

图 2-1

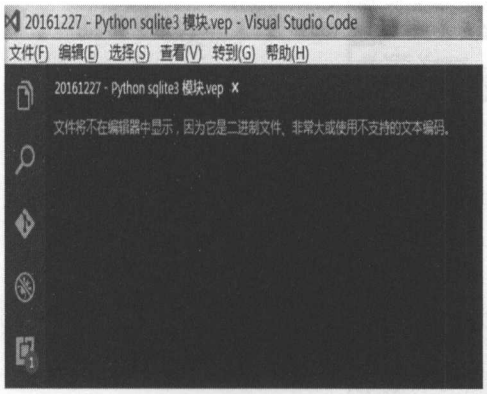


图 2-2

图 2-3

1. sublime3 下载

在 sublime 官网[1]下载相应系统版本的安装包，如图 2-4 所示。

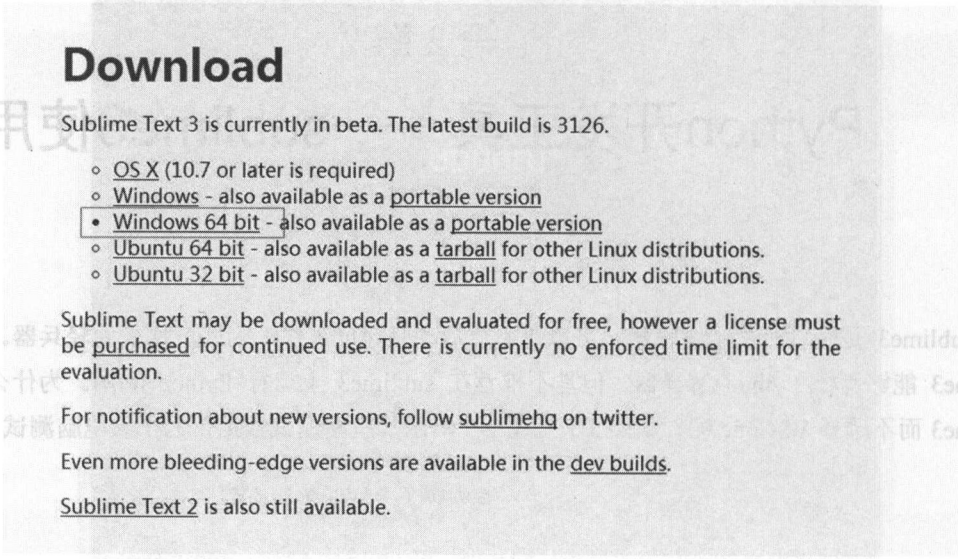


图 2-4

2. sublime 安装

下载好安装包后，直接双击安装，如图 2-5 所示。

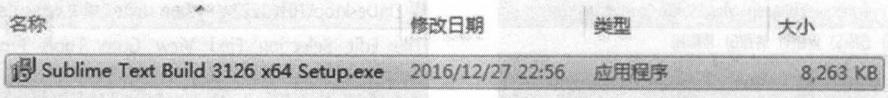


图 2-5

在安装过程中，选择在主菜单上面创建目录，如图 2-6 所示。

1 http://www.sublimetext.com/3/

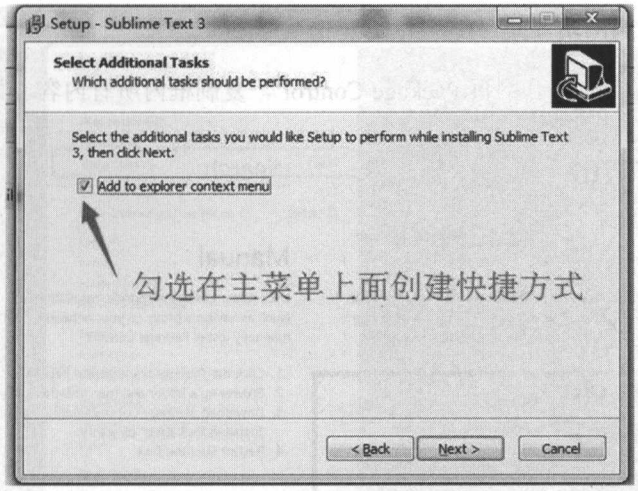


图 2-6

安装完毕后在开始菜单中能看到 sublime3 的快捷方式，如图 2-7 所示。

图 2-7

3. 插件管理工具网站

安装插件管理工具，打开网页 Package Control[1]，复制框内所有内容，如图 2-8 所示。

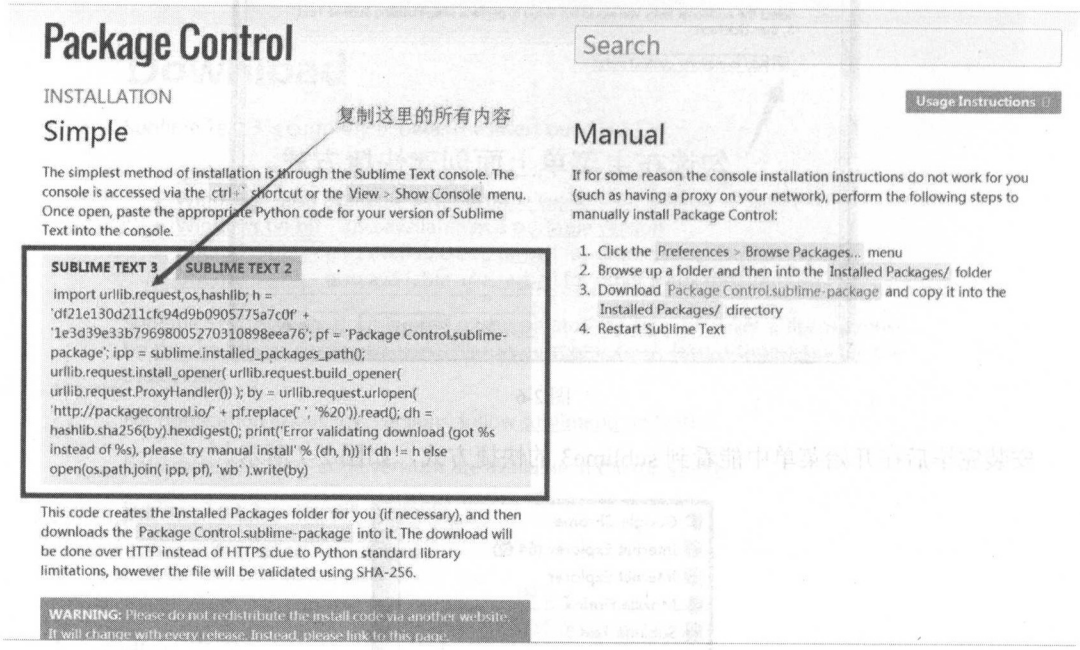


图 2-8

4. 显示控制台

打开 sublime3，按快捷键 ctrl+` 或单击菜单 View > Show Console，打开 Sublime 控制台，如图 2-9 所示。

1 https://packagecontrol.io/installation

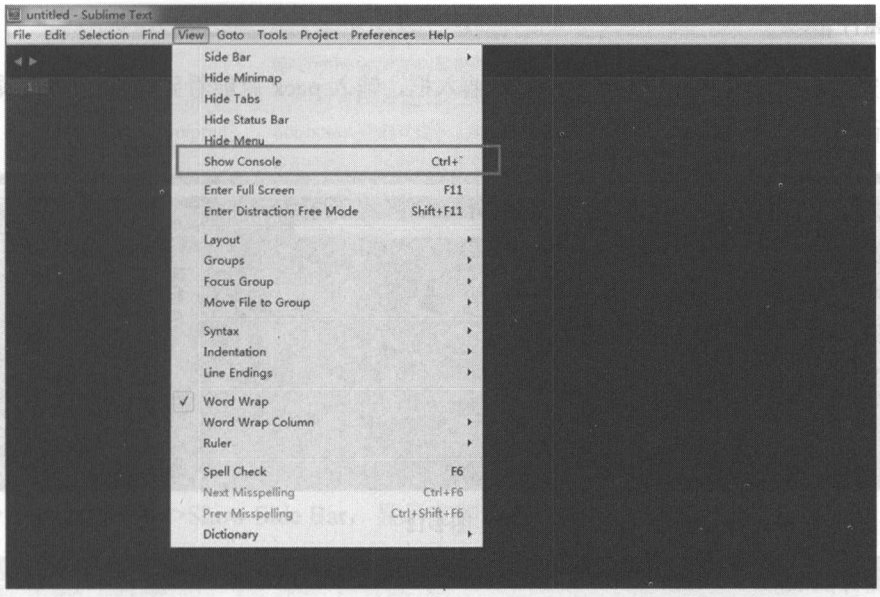


图 2-9

这里的控制台使用了 sublime 内置的 Python，在控制台输入以下代码即可看到使用的 Python 版本，如图 2-10 所示。

```
import sys
sys.version_info
```

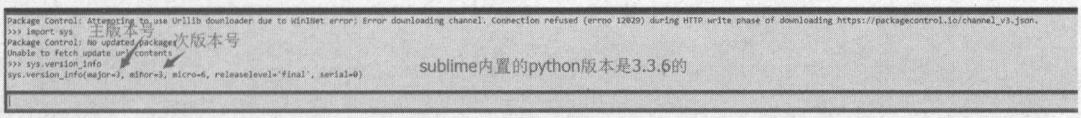


图 2-10

5. 安装插件管理工具

把刚刚复制的代码粘贴到控制台上，按回车键执行即可安装插件管理工具，如图 2-11 所示。

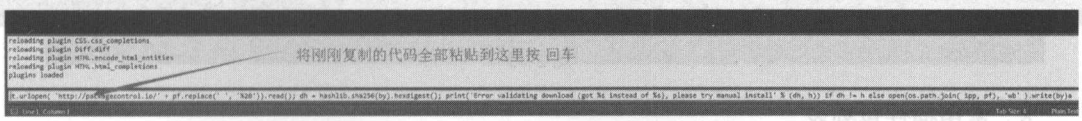


图 2-11

6. 插件管理

安装完成后按 Ctrl+Shift+p 组合键调出输入框，输入 pack 就能看到插件管理的项目了，如图 2-12 所示。

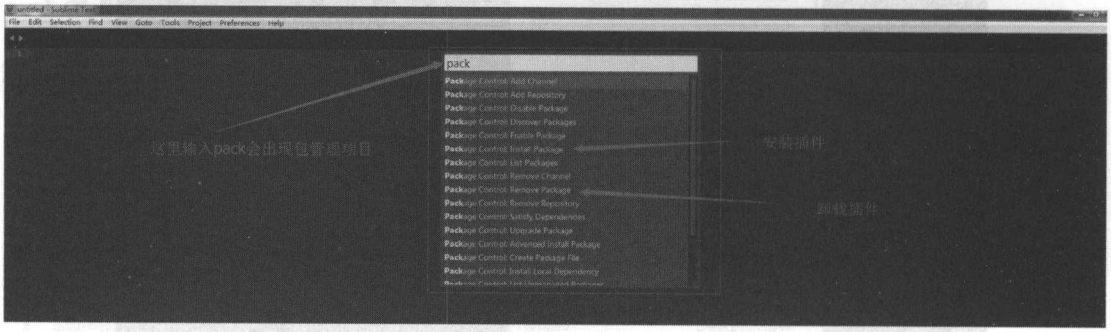


图 2-12

7. 插件安装

用鼠标左键选中 Package Control:Install Package 即可安装插件，如图 2-13 所示。

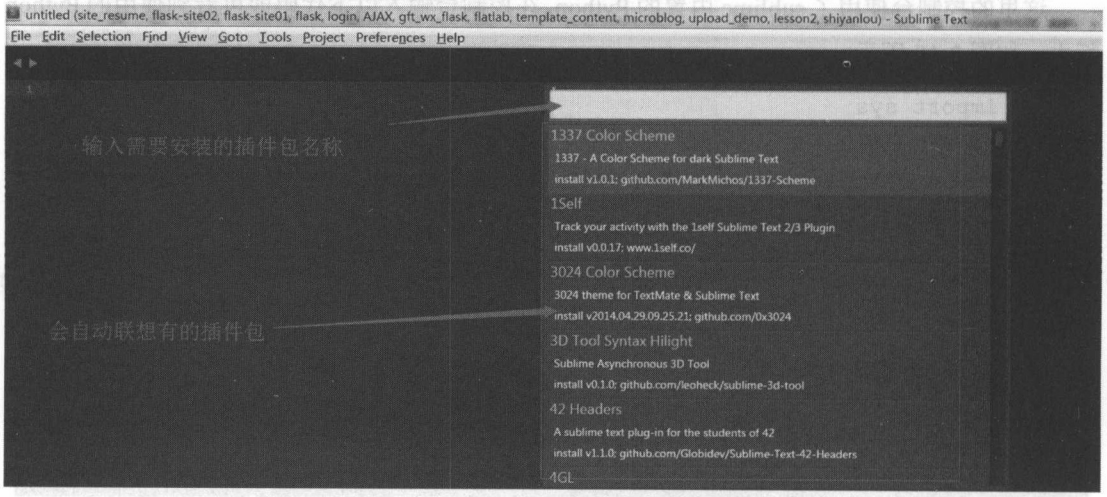


图 2-13

8. 常用插件包列表

安装如图 2-14 所示的 sublime3 常用插件表。

插件包名称	作用
Anaconda	编写python代码的时候提供代码提示
AutoPEP8	按照PEP8规范检查代码格式的插件，帮助语法检查
Bootstrap 3 Snippets	Bootstrap3的自动提示，用于前端编辑
Emmet	html前端代码自动联想补全，输入头几个字母按Tab键会自动补全
Jinja2	Flask开发使用的Jinja模板，编辑html模板时候对Jinja代码自动补全
jquery	html前端ajax自动提示
SideBarEnhancements	侧边栏增强
SFTP	把项目传到远程主机上的插件
SublimeREPL	模拟交互式操作，类似IDLE的操作

图 2-14

9．SideBarEnhancements 包的使用

单击 View>Side Bar>Show Side Bar，显示左侧边栏，如图 2-15 所示。

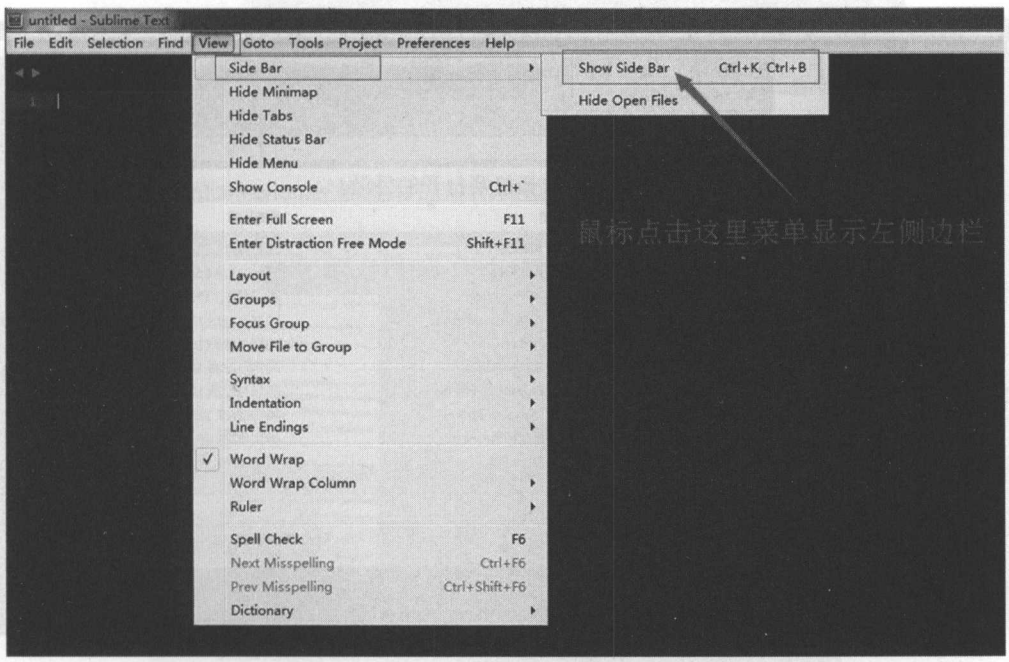


图 2-15

现在左侧边栏打开了，但是在侧边栏上鼠标右键没有菜单显示，解决这个问题的方法是打开一个文件夹，这样鼠标右键才开始生效，如图 2-16 所示。

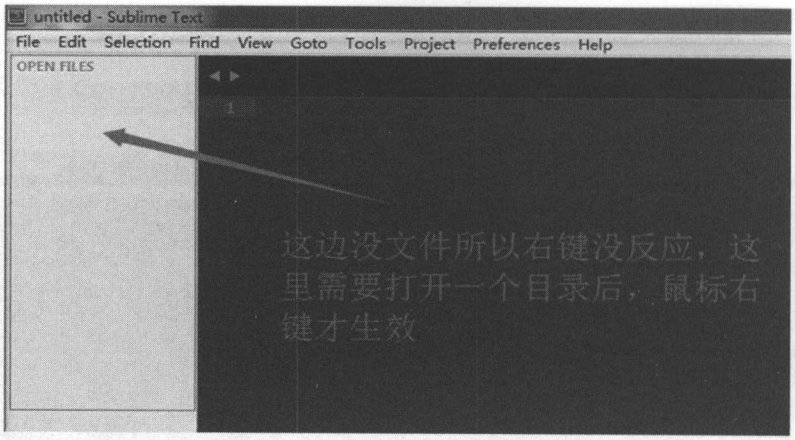


图 2-16

单击 File>Open Folder，打开一个目录，如图 2-17 所示。

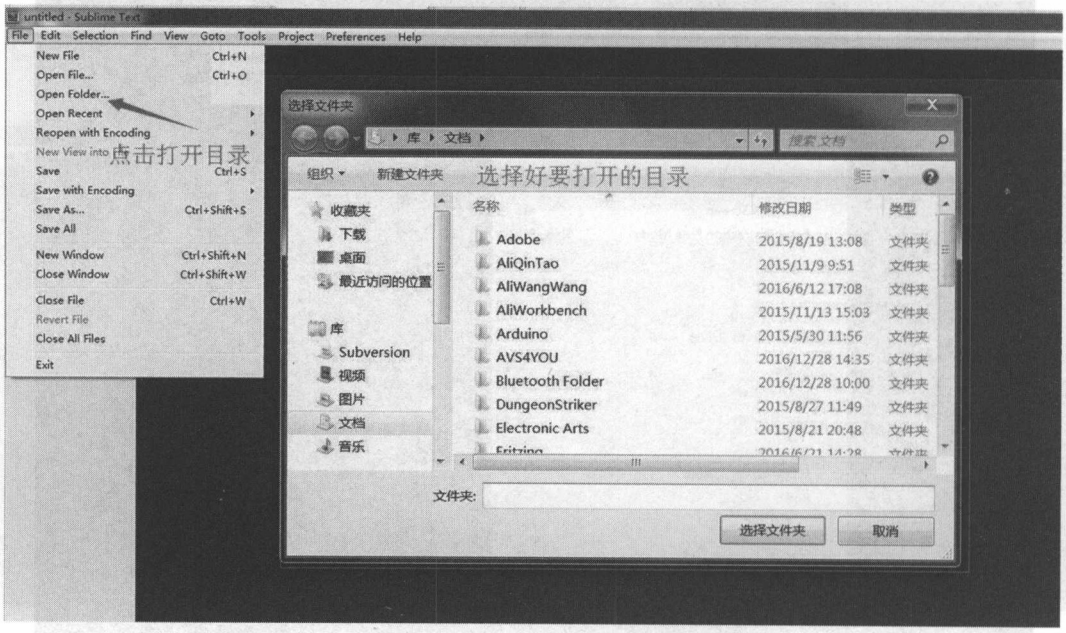


图 2-17

原生 sublime3 菜单如图 2-18 所示。

图 2-18

安装好 SideBarEnhancements 插件后的对比如图 2-19 所示。

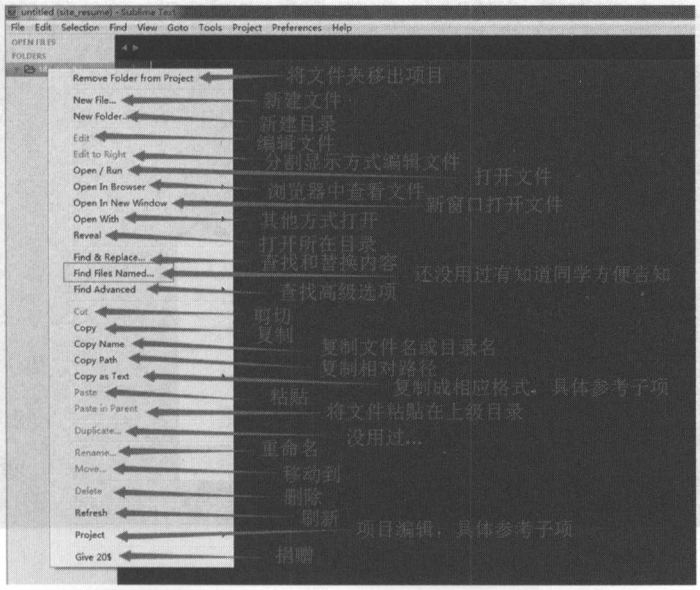


图 2-19

10. AutoPEP8 包使用

菜单使用如图 2-20 所示。

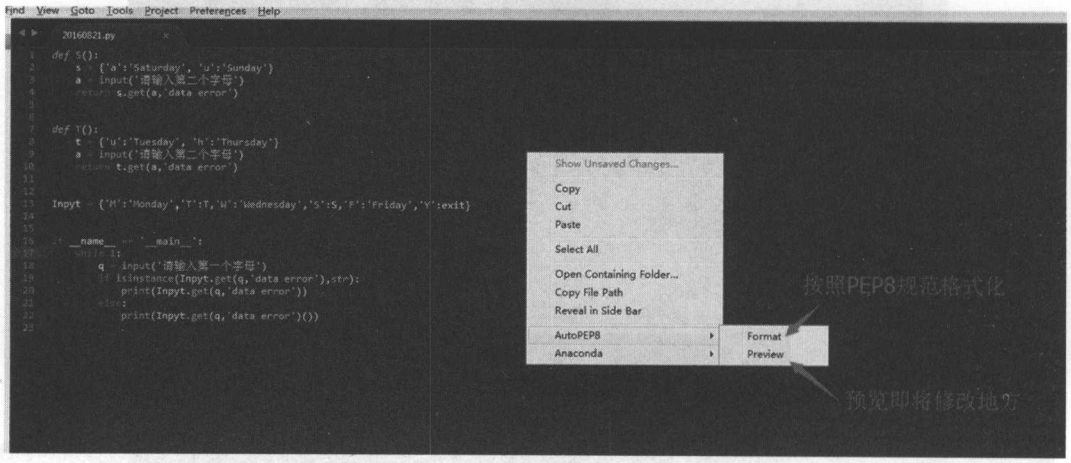


图 2-20

11. Anaconda 包的使用

使用如图 2-21 所示的菜单。

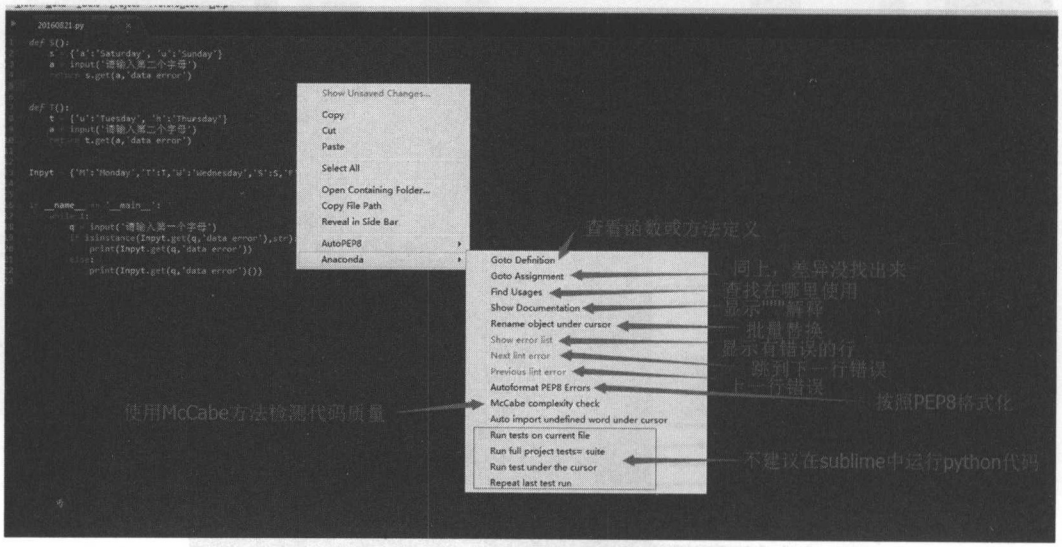


图 2-21

更多使用方法请参考 anaconda 官网[1]。

12．Emmet 包的使用

Emmet 包是自动生成 HTML 代码的插件，用法如下。

新建 HTML 文件，如图 2-22 所示。

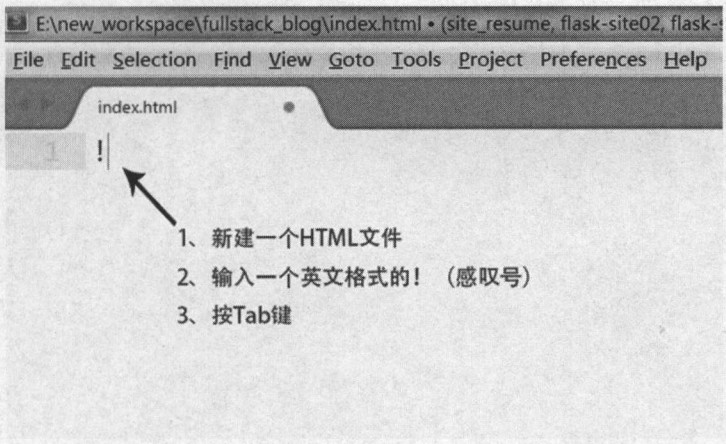


图 2-22

按 Tab 键后会自动生成一个如图 2-23 所示的 HTML 模板。

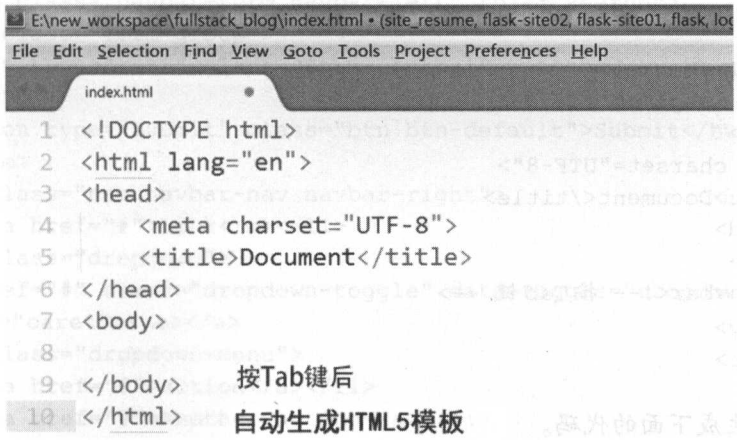


图 2-23

1　http://damnwidget.github.io/anaconda/

其他 HTML 标签可以输入前面命令按 Tab 键生成，更多使用方法请参考 Emmet 官网[1]。

13. Bootstrap 3 Snippets 包的使用

在<body>标签中输入“bs3-xxx”命令可以快速生成 Bootstrap 3 模板，生成 bs3-navbar 导航菜单的方法如图 2-24 所示。

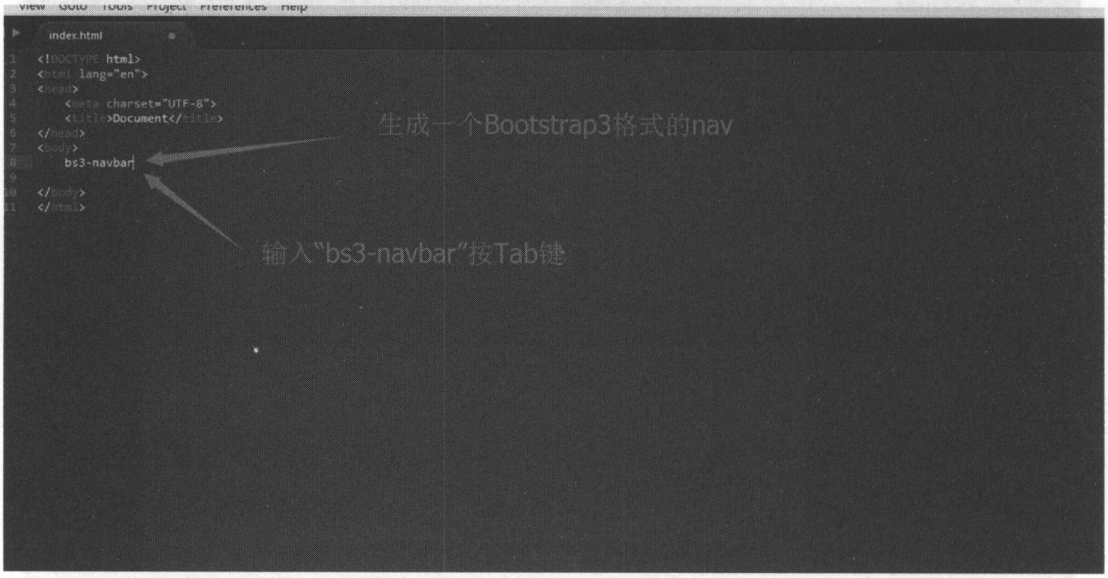


图 2-24

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Document</title>
</head>
<body>
bs3-navbar<!-- 按 Tab 键 -->
</body>
</html>
```

自动生成下面的代码。

```
<!DOCTYPE html>
```

1 http://docs.emmet.io/

```
    <html lang="en">
    <head>
    <meta charset="UTF-8">
    <title>Document</title>
    </head>
    <body>
    <nav class="navbar navbar-default" role="navigation">
    <div class="container-fluid">
    <!-- Brand and toggle get grouped for better mobile display -->
    <div class="navbar-header">
    <button type="button" class="navbar-toggle" data-toggle="collapse" d
ata-target=".navbar-ex1-collapse">
    <span class="sr-only">Toggle navigation</span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    </button>
    <a class="navbar-brand" href="#">Title</a>
    </div>
    <!-- Collect the nav links, forms, and other content for toggling -->
    <div class="collapse navbar-collapse navbar-ex1-collapse">
    <ul class="nav navbar-nav">
    <li class="active"><a href="#">Link</a></li>
    <li><a href="#">Link</a></li>
    </ul>
    <form class="navbar-form navbar-left" role="search">
    <div class="form-group">
    <input type="text" class="form-control" placeholder="Search">
    </div>
    <button type="submit" class="btn btn-default">Submit</button>
    </form>
    <ul class="nav navbar-nav navbar-right">
    <li><a href="#">Link</a></li>
    <li class="dropdown">
    <a href="#" class="dropdown-toggle" data-toggle="dropdown">Dropdown
<b class="caret"></b></a>
    <ul class="dropdown-menu">
    <li><a href="#">Action</a></li>
    <li><a href="#">Another action</a></li>
    <li><a href="#">Something else here</a></li>
    <li><a href="#">Separated link</a></li>
    </ul>
    </li>
    </ul>
```

```
</div><!-- /.navbar-collapse -->
</div>
</nav>
</body>
</html>
```

更多的快捷命令请参考 Bootstarp 3 Snippets[1]插件。

14. Jinja2 包的使用

Jinja 模板是在 flaska 或者 django 开发中经常使用的前端模板。将 HTML 模板切换成 HTML 带 Jinja 模板的方法有两种，如图 2-25 所示。

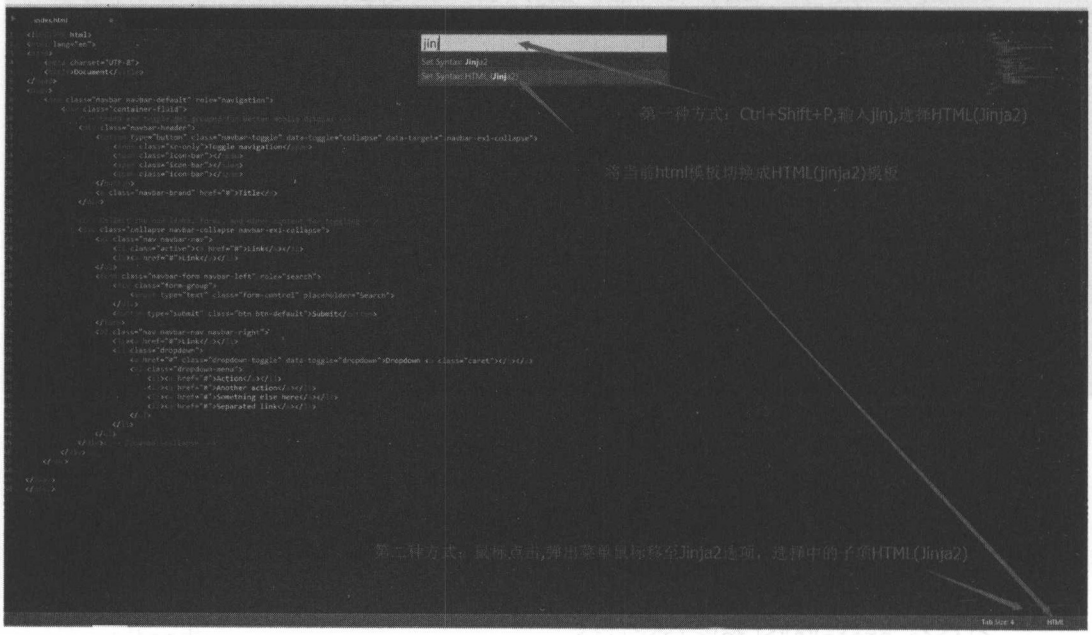


图 2-25

具体 Jinja2 模板的使用方法请参考 Jinja2 官网[2]。

15. jQuery 包的使用

jQuery 插件包是自动提示 jQuery 的代码。

---

1 https://github.com/JasonMortonNZ/bs3-sublime-plugin#input-fields-form-fields

2 http://jinja.pocoo.org/docs/dev/

在</body>和</html>中间输入 script，按 Tab 键会自动生成<script>标签，在<script>标签中输入 j 会自动弹出菜单，选择 Ajax 则会生成完整的 Ajax 模板，如图 2-26 所示。

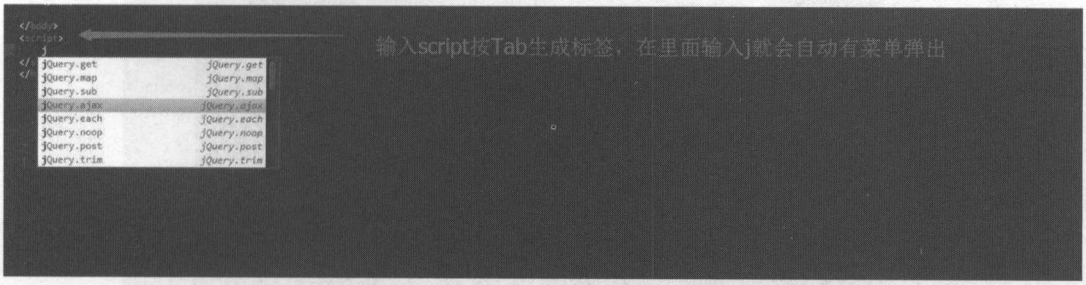


图 2-26

补全后的 jQuery 代码如图 2-27 所示。

```
</body>
<script>
    jQuery.ajax({
      url: '/path/to/file',
      type: 'POST',
      dataType: 'xml/html/script/json/jsonp',
      data: {param1: 'value1'},
      complete: function(xhr, textStatus) {
        //called when complete
      },
      success: function(data, textStatus, xhr) {
        //called when successful
      },
      error: function(xhr, textStatus, errorThrown) {
        //called when there is an error
      }
    });

</script>
</html>a
```

图 2-27

16. SFTP 包的使用

安装好 SFTP 插件后，第一次使用时需要生成配置文件，即在当前目录生成一个 sftp-config.json 配置文件，如图 2-28 所示。

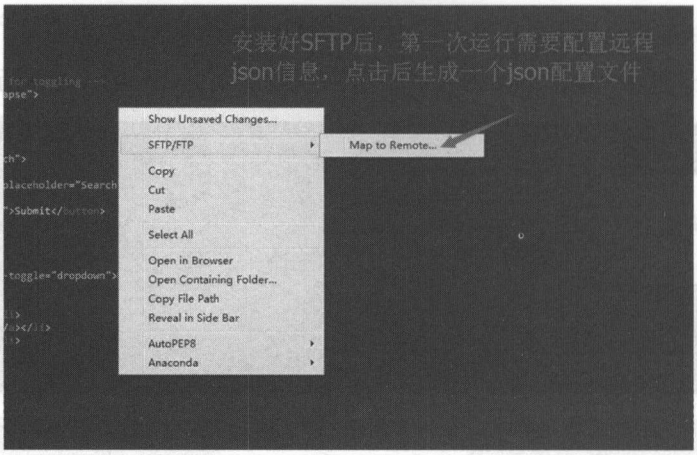


图 2-28

具体配置内容如图 2-29 所示。

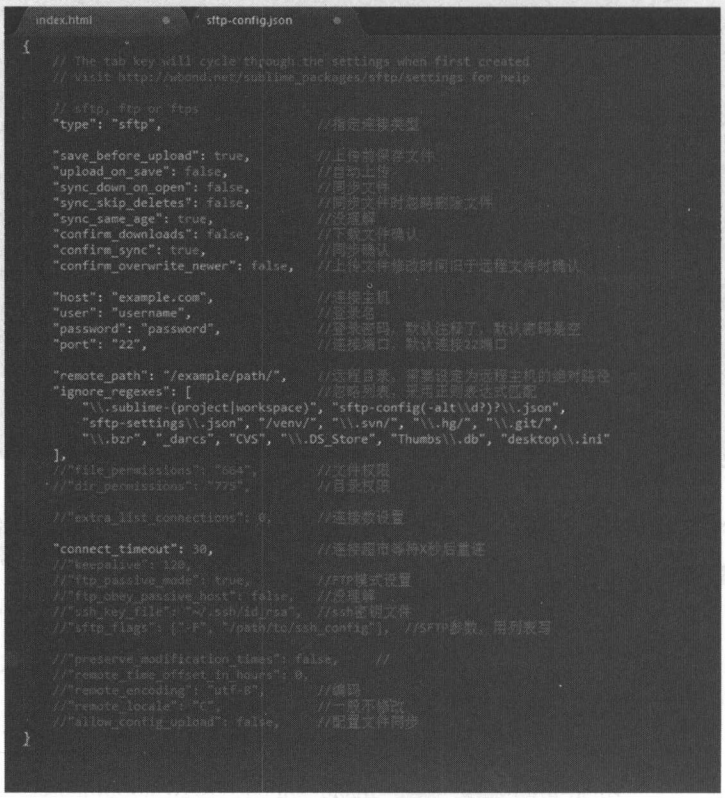


图 2-29

使用 SFTP 上传当前文件，如图 2-30 所示。

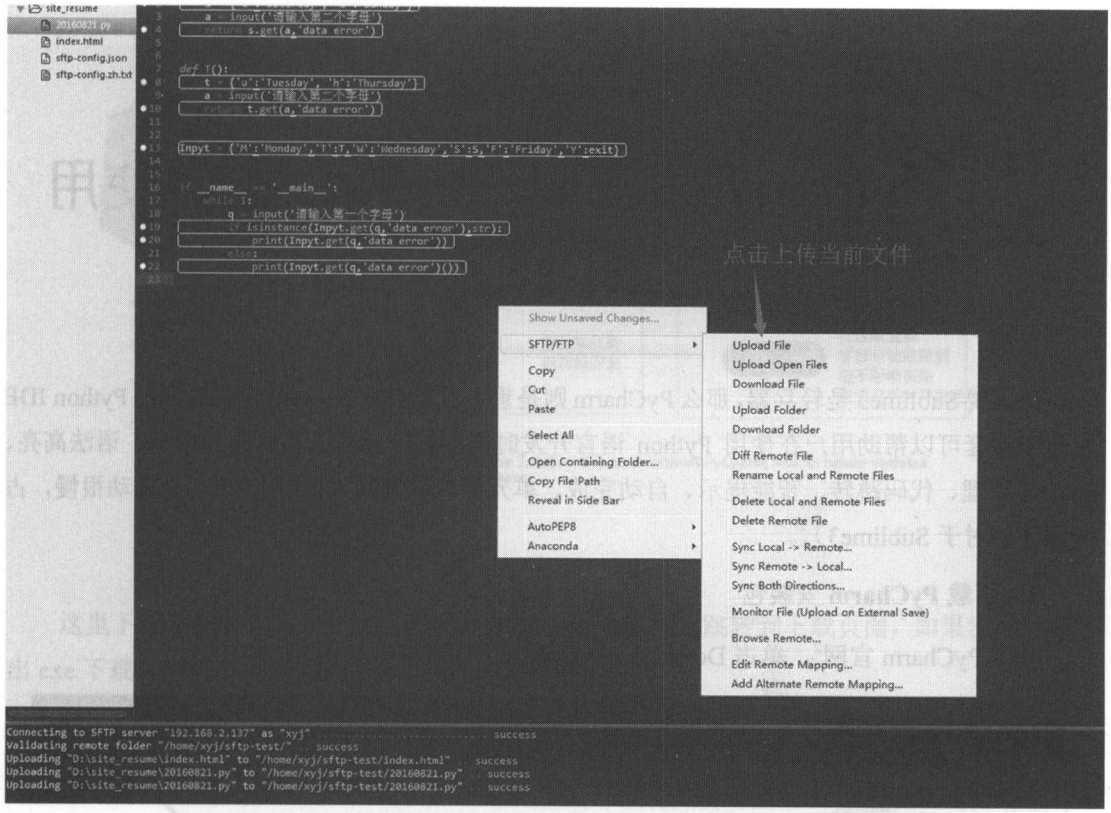


图 2-30

第3章
Python开发工具——PyCharm使用

如果说Sublime3是轻兵器，那么PyCharm则是重型武器。PyCharm是一个强大的Python IDE，包含一整套可以帮助用户在使用 Python 语言开发时提高其效率的工具，比如调试、语法高亮、Project管理、代码跳转、智能提示、自动完成、单元测试以及版本控制。[1]不足是启动很慢，占内存（相对于Sublime3）。

1. 下载 PyCharm 安装包

打开 PyCharm 官网[2]，单击 Download 按钮进入下载页面，如图 3-1 所示。

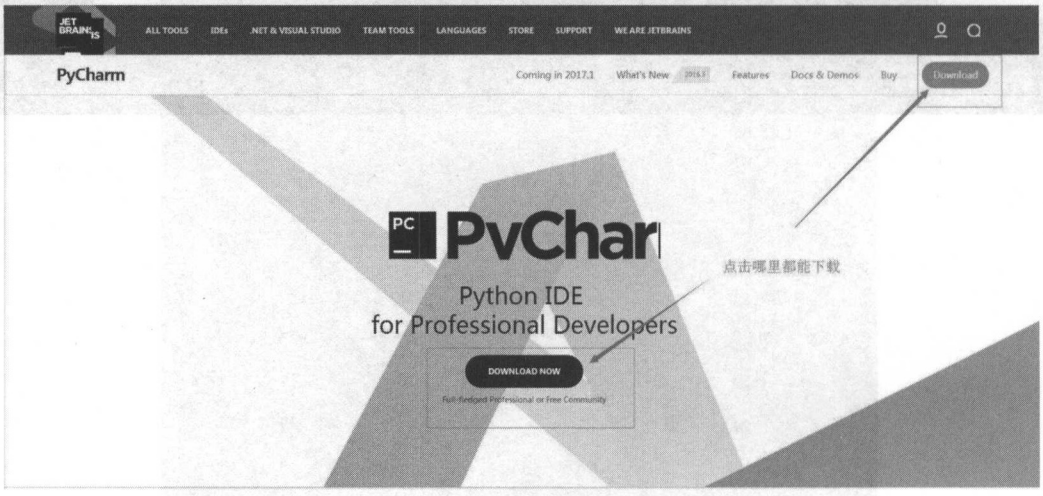


图 3-1

1 http://baike.baidu.com/item/pyCharm/8143824?fr:alddin

2 http://www.jetbrains.com/pycharm/

选择对应的系统版本下载安装包，如图 3-2 所示。

图 3-2

这里下载专业版，单击专业版下面的 DOWNLOAD 按钮跳转到下载页面，如果没有自动弹出 exe 下载，则单击如图 3-3 所示的红框内的链接。

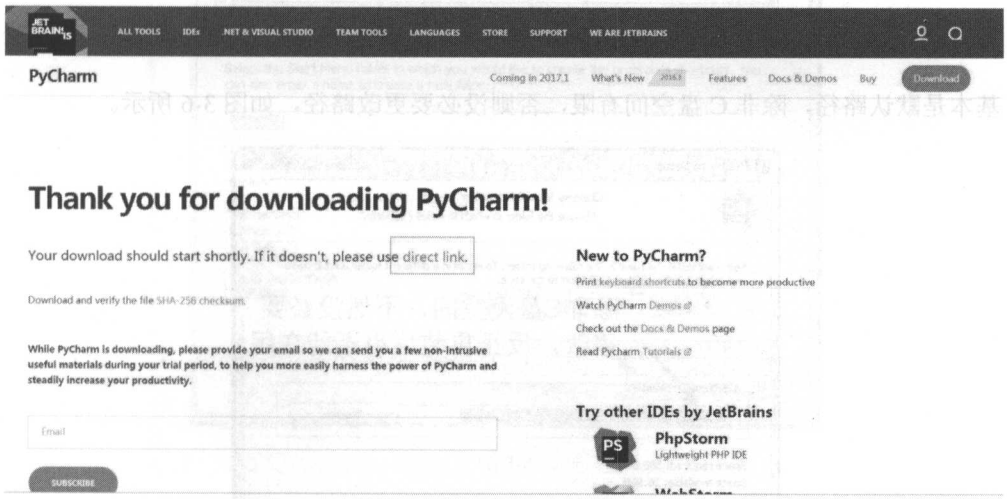


图 3-3

2. PyCharm 安装

下载安装包后，双击安装，如图 3-4 所示。

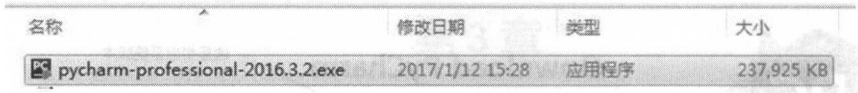


图 3-4

安装步骤如图 3-5 所示。

图 3-5

基本是默认路径，除非 C 盘空间有限，否则没必要更改路径，如图 3-6 所示。

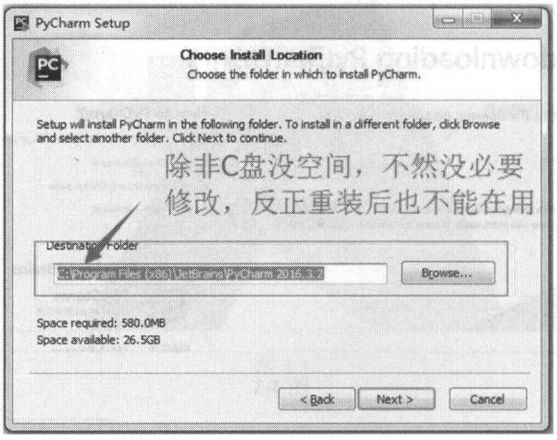


图 3-6

选择在桌面创建快捷方式，关联所有 Python 源文件，如图 3-7 所示。

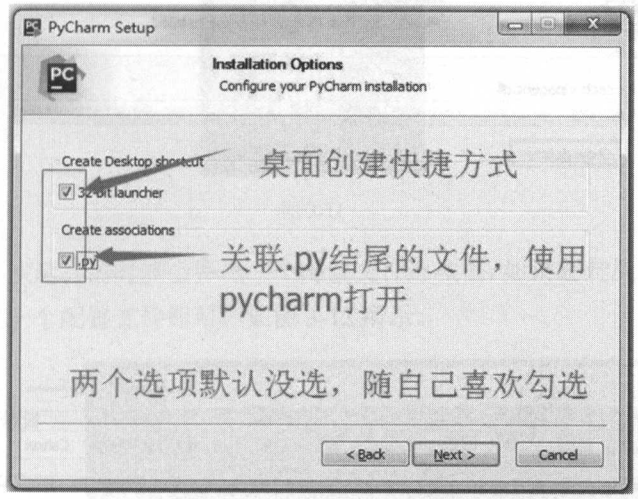


图 3-7

在主菜单创建目录（默认选择），单击 Install 按钮，如图 3-8 所示。

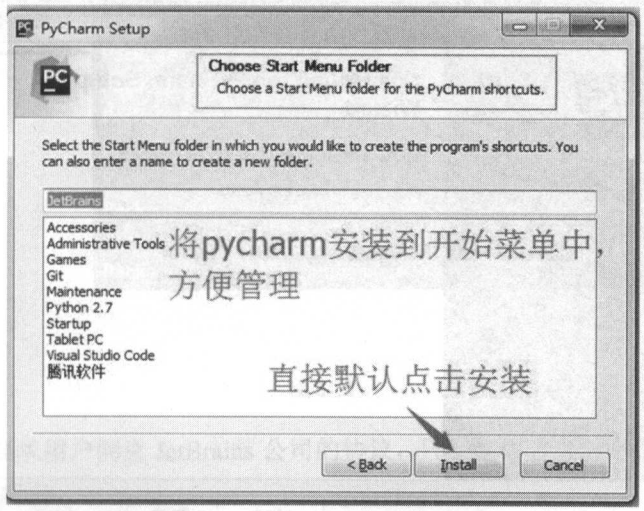


图 3-8

安装过程如图 3-9 所示。

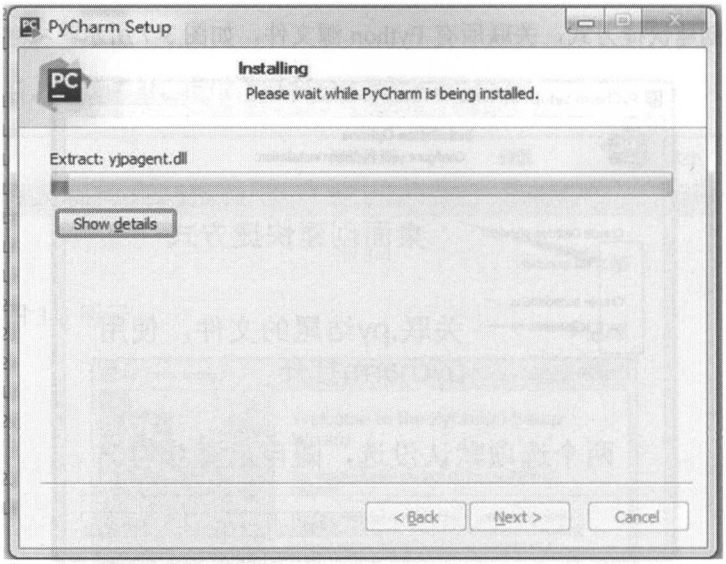


图 3-9

安装完成，如图 3-10 所示。

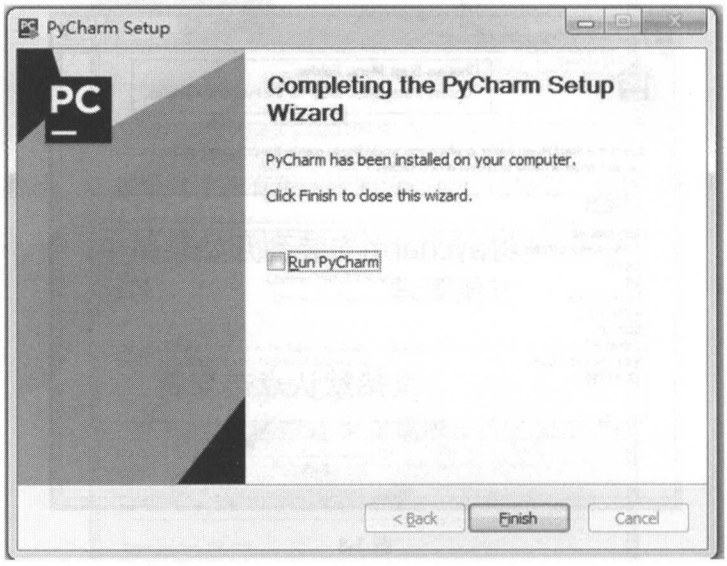


图 3-10

3．激活软件

安装完成后在桌面会生成一个快捷方式，双击即可运行 PyCharm，如图 3-11 所示。

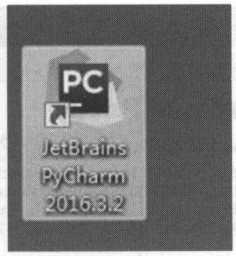


图 3-11

PyCharm 在第一次运行的时候会弹出一个配置文件选项框，如果之前从未使用过 PyCharm，则按照默认选择新建一个配置文件即可，如图 3-12 所示。

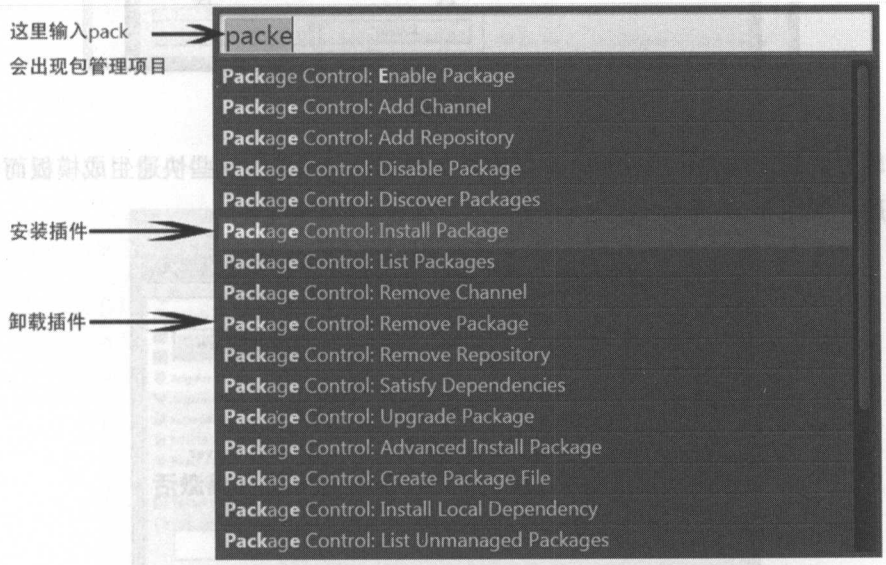


图 3-12

第一次使用会要求用户同意 JetBrains 公司的协议，只有选择 Accept 按钮（同意）才能使用该软件，如图 3-13 所示。

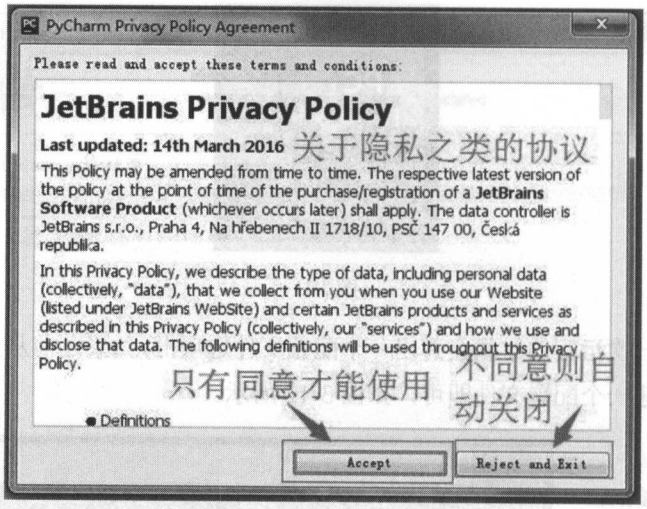


图 3-13

激活软件，建议购买正版软件，社区版是免费的，只是缺少一些快速生成模板而已，基本使用没问题，如图 3-14 所示。

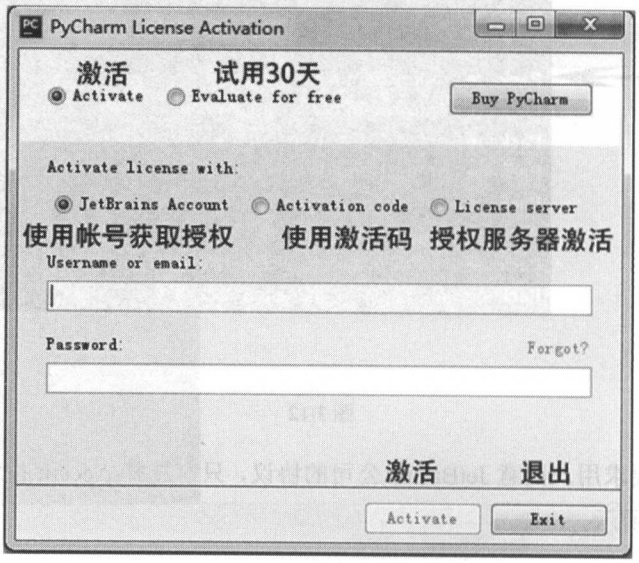


图 3-14

4．创建第一个项目

这里选择新建项目，如图 3-15 所示。

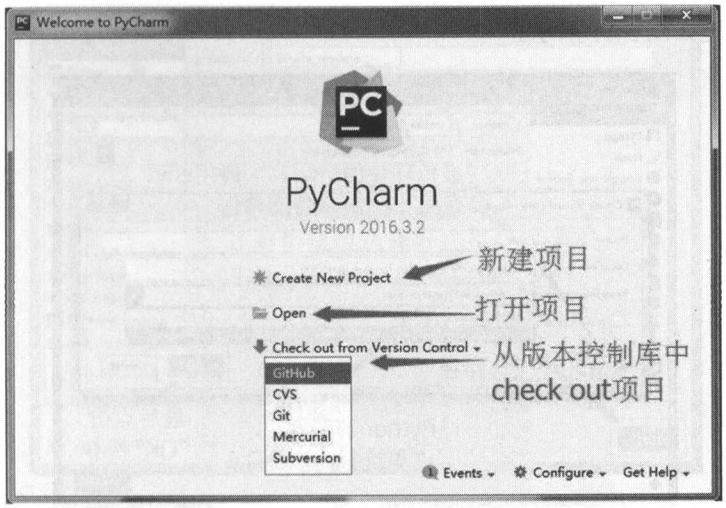


图 3-15

项目模板、项目保存位置、项目解释器选择如图 3-16 所示。

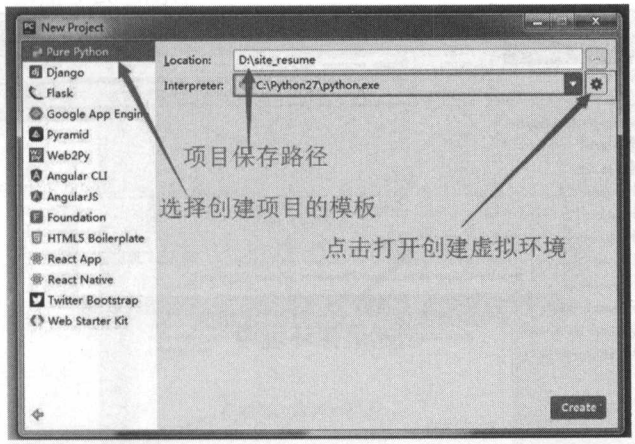


图 3-16

创建 Python 虚拟环境，虚拟环境是创建一个“干净”的 Python 的运行环境。为什么要用虚拟环境，而不直接使用系统原生呢？如果只是开发一个项目在单一服务器上运行当然没问题，但在开发多个项目后，导出相关引用的第三方包的时候就会发现，在系统 Python 下的项目中有很多第三方包，有些是这个项目的，有些不是这个项目的。而使用虚拟环境则可以给每一个项目配置一个虚拟环境，这样这个项目使用的第三方包就很明确，迁移到其他机器上时只需导出相关依赖清单 requirements 就可以了。

创建虚拟环境如图 3-17 所示。

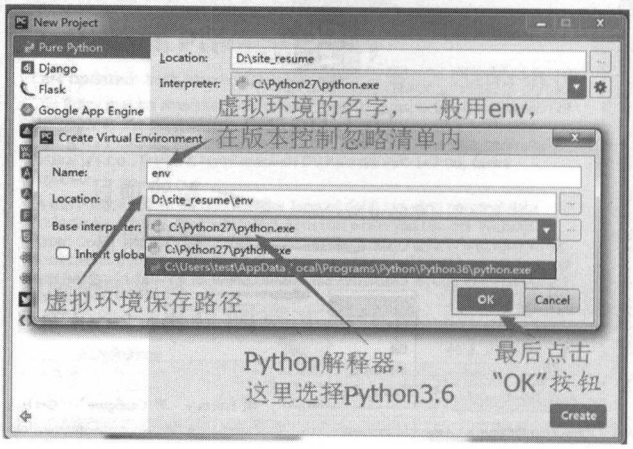


图 3-17

虚拟环境配置如图 3-18 所示。

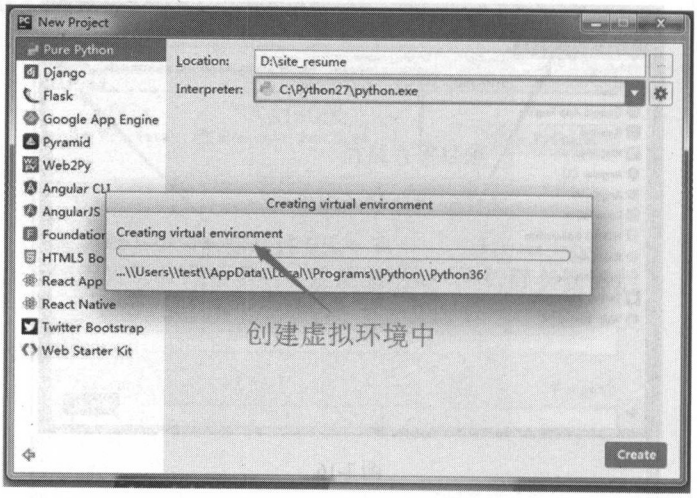


图 3-18

配置完成后就可以生成项目框架了，如图 3-19 所示。

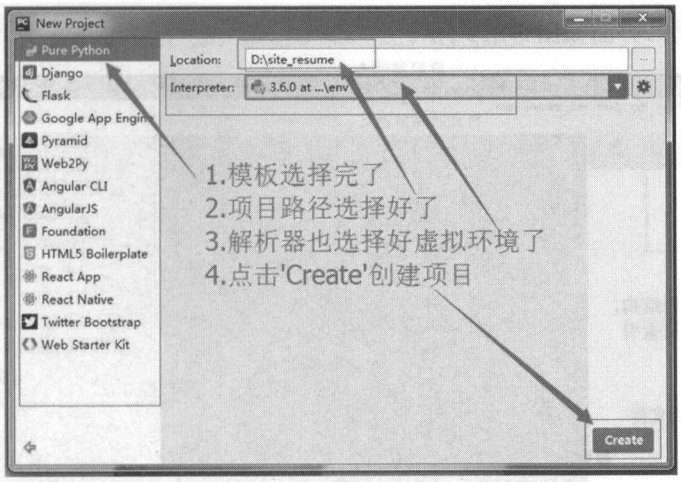


图 3-19

5. 初识 PyCharm 界面

关闭每日提示，如图 3-20 所示。

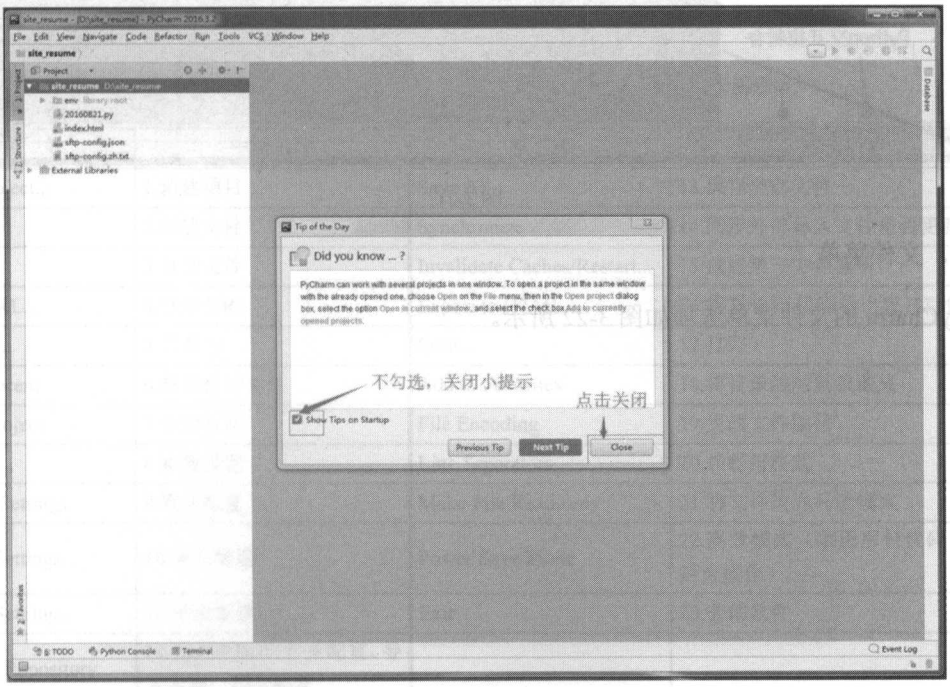


图 3-20

PyCharm 启动后界面如图 3-21 所示。

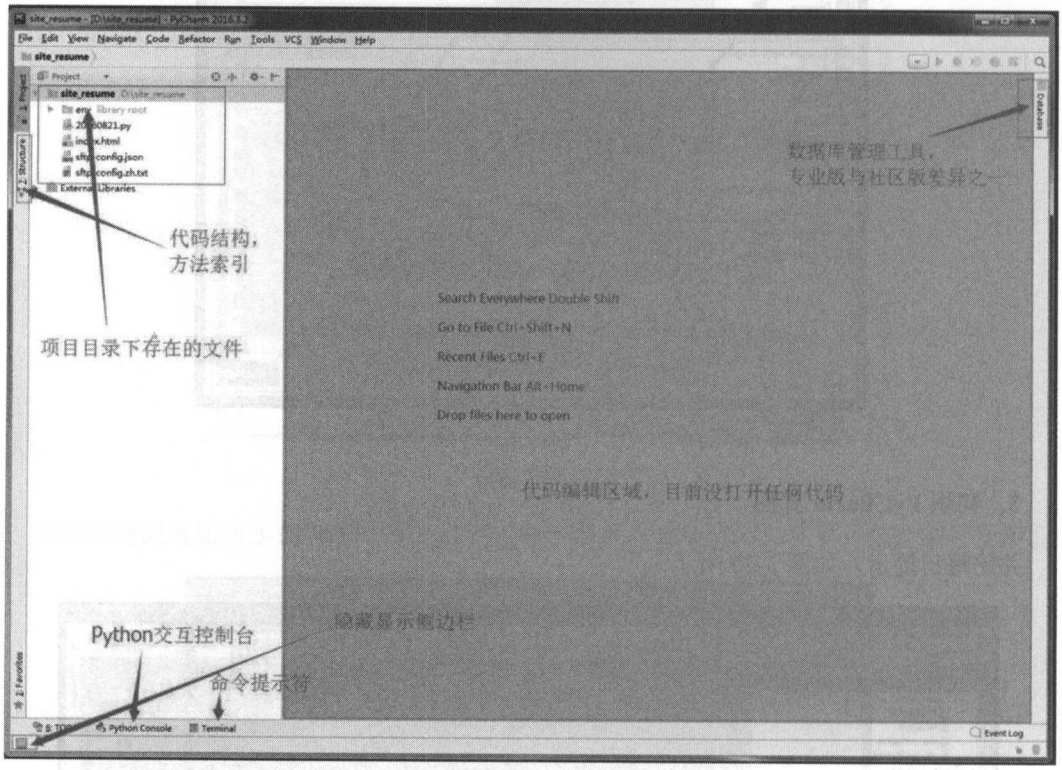


图 3-21

6. 文件菜单

PyCharm 的文件菜单选项如图 3-22 所示。

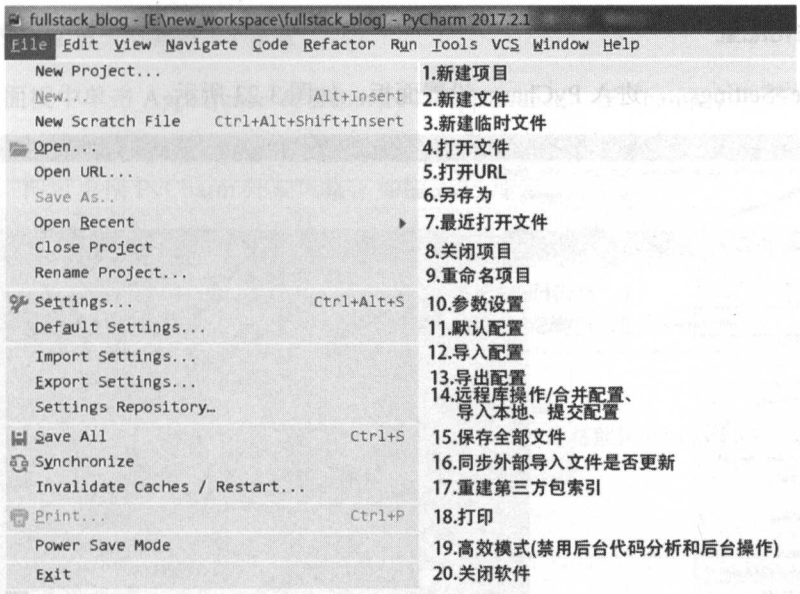


图 3-22

文件菜单选项的中文名称如表 3-1 所示。

表 3-1

原　文	中文名称	原　文	中文名称
New Project...	1.新建项目	Save All	13.保存全部文件
New...	2.新建文件	Synchronize	14.同步外部导入文件是否更新
Open...	3.打开文件	Invalidate Caches/Restart...	15.重建第三方包索引
Open URL...	4.打开 URL	Export to HTML...	16.将 Python 代码导出成 HTML 文件
Save As...	5.另存为	Print...	17.打印
Open Recent	6.最近打开文件	Add to Favorites	18.将目录添加到收藏夹
Close Project	7.关闭项目	File Encoding	19.更改文件编码
Settings...	8.参数设置	Line Separators	20.换行符格式
Default Settings...	9.默认配置	Make File Read-only	21.将文件改为只读模式
Import Settings...	10.导入配置	Power Save Mode	22.高效模式（禁用后台代码分析和后台操作）
Export Settings...	11.导出配置	Exit	23.关闭软件
Settings Repository...	12.远程库操作/合并配置、导入本地、提交配置		

7. 个性化配置

单击 File>Settings...，进入 PyCharm 设置面板，如图 3-23 所示。

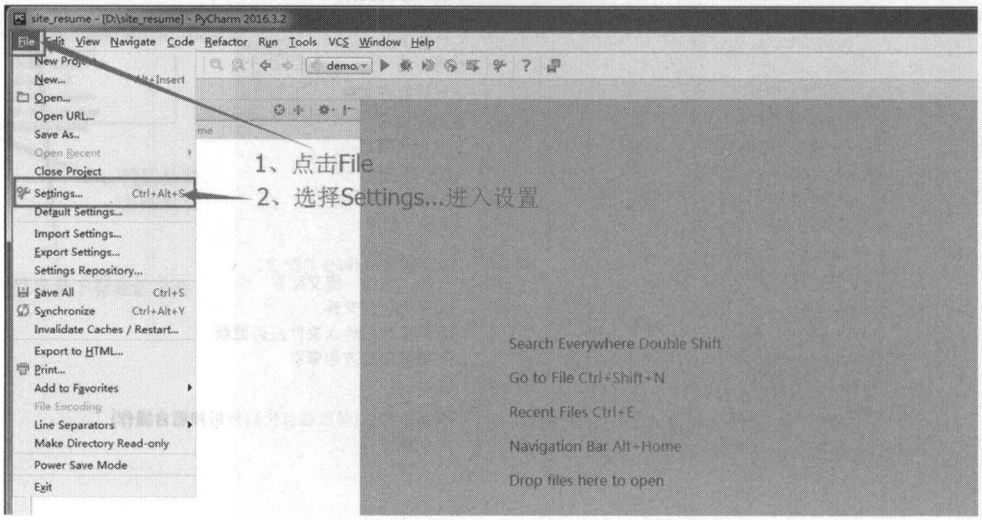


图 3-23

设置面板功能如图 3-24 所示。

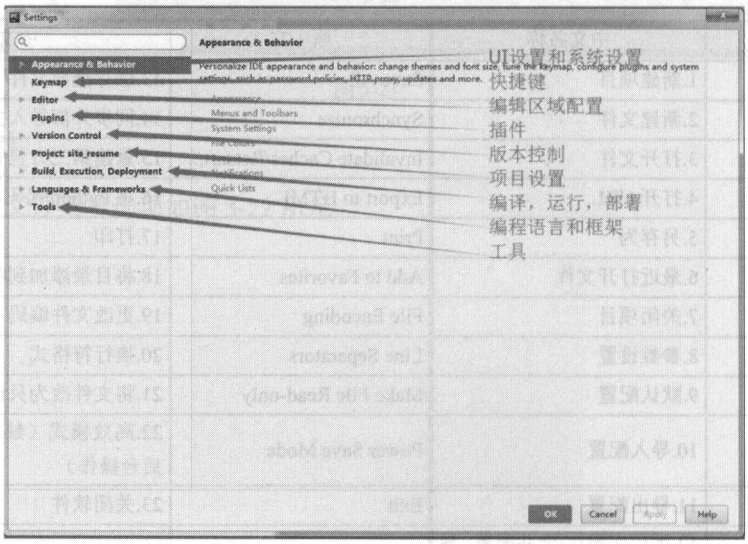


图 3-24

8. 个性化配置——黑炫 UI 设置

在设置面板中单击 Appearance & Behavior > Appearance，在 Theme 中选择 Darcula 主题，勾选 Override default fonts by (not recommended)，将默认字体切换成微软雅黑等宽字体，单击 Apply 按钮，即可更换 PyCharm 外观风格，如图 3-25 所示。

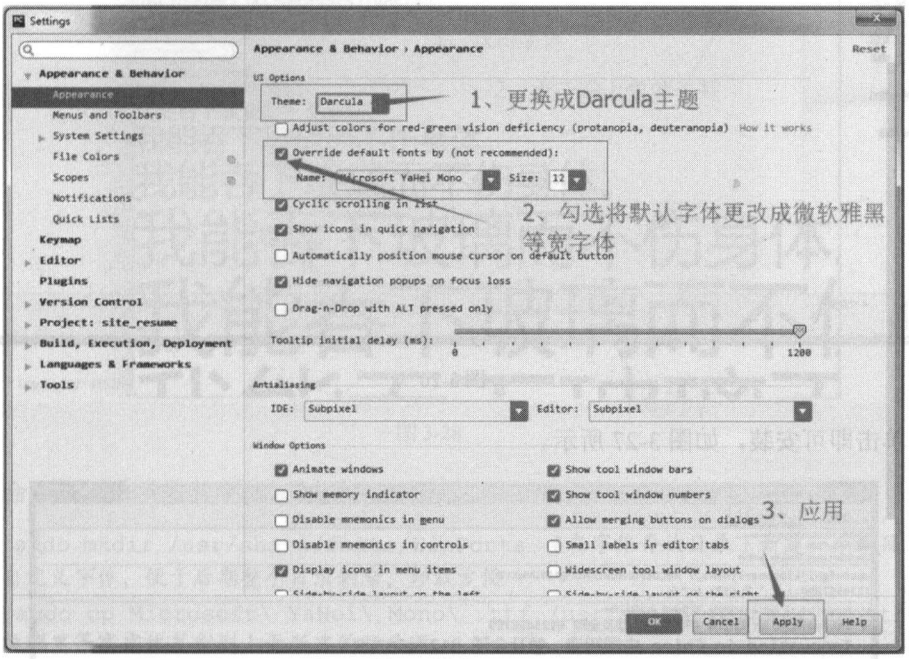


图 3-25

9. 个性化配置——字体设置

前置下载字体，推荐使用微软雅黑等宽字体 Microsoft YaHei Mono。自行用搜索查找 Microsoft YaHei Mono 下载。

（1）Windows 下安装字体

① 找到下载好的字体文件，双击打开，如图 3-26 所示。

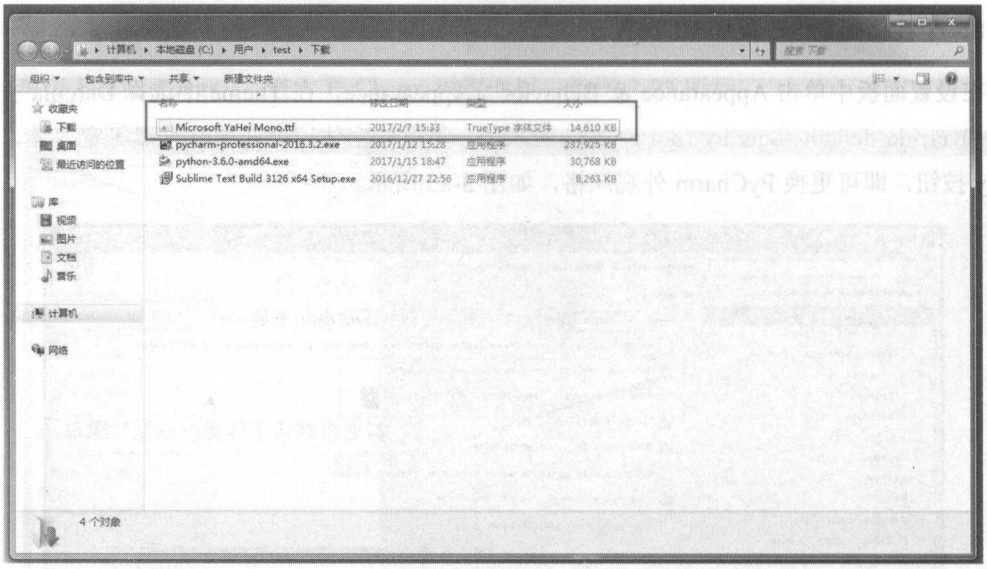


图 3-26

② 单击即可安装，如图 3-27 所示。

图 3-27

（2）Linux 下安装字体

① 在图形化界面上面双击字体即可自动安装，如图 3-28 所示。

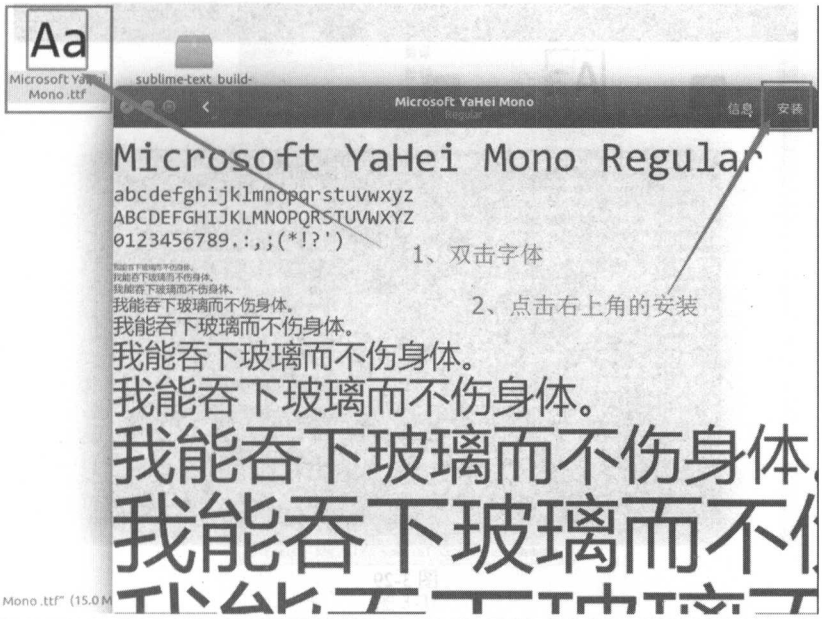


图 3-28

② 在命令提示符下安装字体，如图 3-29 所示。

```
$ sudo mkdir /usr/share/fonts/WinFonts  #在字体存放目录下新建一个新目录用来
#存放自定义字体，便于后期整个目录删除，卸载方便
$ sudo cp Microsoft\ YaHei\ Mono\ .ttf /usr/share/fonts/WinFonts/
#将微软雅黑等宽字体复制到上面新建的目录下
$ cd /usr/share/fonts/WinFonts/  #切换到自定义字体存放目录
$ sudo mkfontscale  #创建 fonts.scale 文件，该文件用来控制字体旋转缩放
$ sudo mkfontdir  #fonts.dir 文件，该文件用来控制字体粗斜
$ sudo fc-cache -fv  #建立字体缓存信息
```

单击图 3-29 中的 Editor>Colors & Fonts 选项，显示如图 3-30 所示。

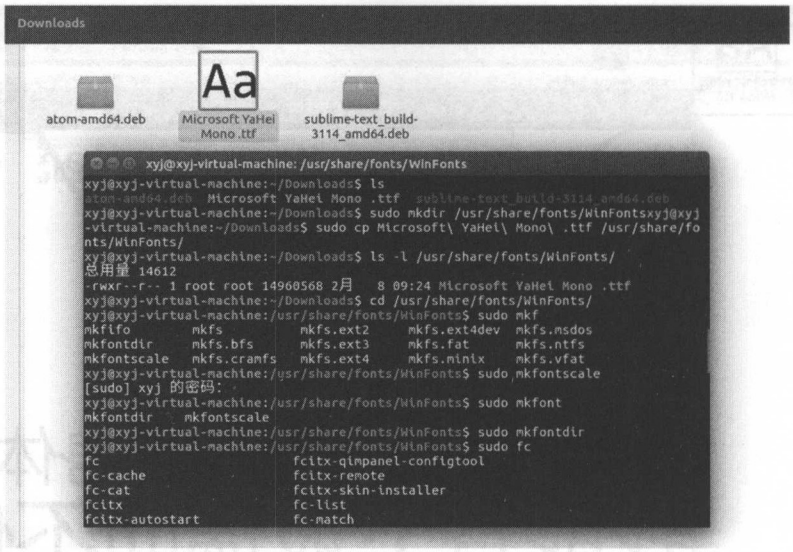


图 3-29

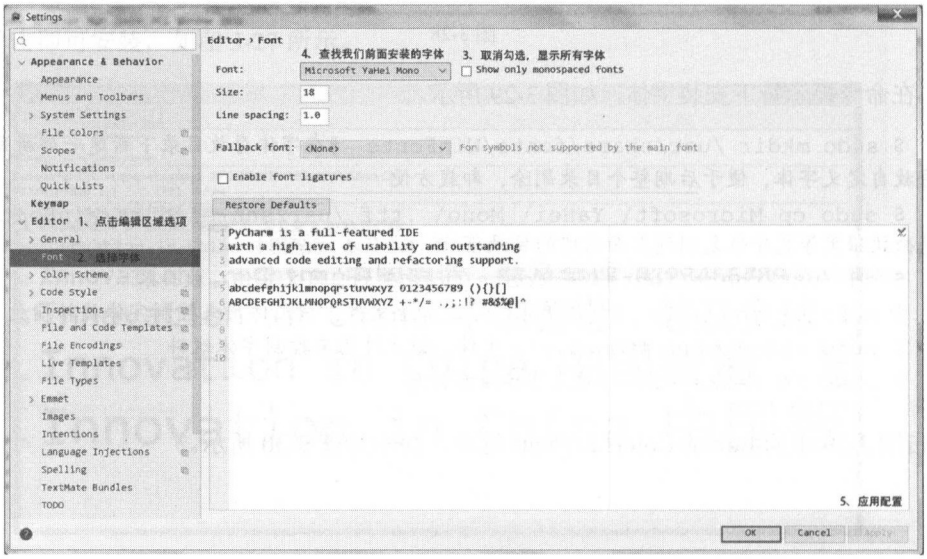


图 3-30

10. 编辑菜单

编辑（Edit）菜单如图 3-31 所示。

图 3-31

具体含义如表 3-3 所示。

表 3-3

原　　文	中文名称	原　　文	中文名称
undo	撤销	select all	全选
redo	重做	extend selection	扩展选择内容，即选择整段代码
cut	剪切	shrink selection	选择内容缩减至最后一个单词
copy	复制	join lines	连接行
copy path	复制当前文件所在路径	fill paragraph	填充段落
copy as plain text	复制成纯文本	duplicate line	复制行
copy reference	复制引用位置	indent selection	缩进
paste	粘贴	unindent line or selection	取消缩进
paste from history	从剪切板历史中选择粘贴	toggle case	字面意思是：切换案例，但实际是把代码转换成大写字母
paste simple	简单粘贴	conver indents	转换缩进
delete	删除	next parameter	
find	查找功能	previous parameter	
macros	宏功能	encode XML/HTML special characters	
colum selection mode	开启列选择模式，即可以多行复制	edit as table	显示成表格模式编辑

11. 视图菜单

视图菜单（View）主要显示各种快捷件的功能，如图 3-32 所示。

图 3-32

具体含义如表 3-4 所示。

表 3-4

原　文	中文名称	原　文	中文名称
Tool Windows	工具窗口	Quick Switch Scheme...	快速切换样式
Quick Definition	自定义快捷键	Toolbar	工具栏
Quick documentation	快速文档	Tool Buttons	工具按键
Parameter Info	参数信息	Status Bar	状态栏
Context Info	说明信息	Navigation Bar	导航栏
Recent Files	最近打开过的文件	Active Editor	编辑区域功能选项
Recently Changed Files	最近更改的文件	Bidi Text Direction	Bidi 文字对齐
Recent changes	最近修改过的记录	Enter Presentation Mode	进入演示模式
Compare With..	比较	Enter Distraction Free Mode	无干扰模式，实际把控件边框隐藏了
Compare with Clipboard	与剪贴板中的内容对比	Enter Full Screen	全屏

12. 导航菜单

导航菜单（Navigate）如图 3-33 所示。

图 3-33

具体含义如表 3-5 所示。

表 3-5

原　文	中文名称	原　文	中文名称
Class...	类	Implementation(s)	实现
File...	文件	Type Declaration	类型声明
Symbol...	符号	Super Method	超类方法
Custom Folding...	自定义折叠	Related Symbol...	相关符号
Line...	线	File Structure	文件结构
Back	返回	File Path	文件路径

续表

原　文	中文名称	原　文	中文名称
Forward	向前	Type Hierarchy	类型层次结构
Last Edit Location	最后一次编辑位置	Method Hierarchy	方法层次结构
Next Edit Location	下一个编辑位置	Call Hierarchy	调用层次结构
Bookmarks	书签	Next Highlighted Error	高亮显示下一个错误
Select In...	选择	Previous Highlighted Error	高亮显示上一个错误
Jump to Navigation Bar	转跳到导航栏	Next Method	下一个方法
Declaration	声明	Previous Method	上一个方法

13. 代码菜单

代码菜单（Code）如图 3-34 所示。

图 3-34

具体含义如表 3-6 所示。

表 3-6

原　文	中文名称	原　文	中文名称
Override Methods...	重写方法	Optimize Imports	优化项目导入
Implement Methods...	实现方法	Rearrange Code	重新排列代码
Generate...	生成	Move Statement down	声明下移
Surround With...	添加代码嵌套	Move Statement Up	声明上移
Unwrap/Remove...	删除代码嵌套	Move Element Left	元素左移
Completion	代码智能提示	Move Element Right	元素右移
Folding	折叠代码	Move Line Down	行下移
Insert Live Template...	插入模板	Move Line Up	行上移
Surround with Live Template...	嵌套模板	Inspect Code...	检查代码
Comment with Line Comment	行注释	Code CleanUp	代码清理
comment with Block Comment	块注释	Run Inspection by Name...	按名称运行检验
Reformat Code	使用 PEP8 风格格式化代码	Configure Current File Analysis...	分析当前文件配置
Show Reformat File Dialog	显示格式化文本对话框	View Offline Inspection Results...	查看离线检验结果
Auto-Indent Lines	自动行缩进	Locate Duplicates...	检查重复

14. 重构菜单

重构菜单（Refactor）如图 3-35 所示。

图 3-35

具体含义如表 3-7 所示。

表 3-7

原　文	中文名称	原　文	中文名称
Refactor This...	重构当前类	Extract	提取
Rename...	重命名	Inline...	嵌入
Change Signature...	更改签名	Pull Members Up...	拉取类成员
Move...	移动	Push Members Down...	推送类成员
Copy...	复制	Convert to Python Package	转换成 Python 包
Safe Delete...	安全删除	Convert to Python Module	转换成 Python 模块

15. 运行菜单

运行菜单（Run）如图 3-36 所示。

图 3-36

具体含义如表 3-8 所示。

表 3-8

原　文	中文名称	原　文	中文名称
Run 'demo'	运行‘demo’	Step Into My Code	单步执行（与上面的区别就是跳过系统代码）
Debug 'demo'	调试‘demo’	Force Step Into	强制单步执行
Run 'demo' with Coverage	使用 Coverage 运行‘demo’	Smart Step Into	智能单步执行
Profile 'demo'	执行分析‘demo’（分析脚本内部运行情况）	Step Out	单步跳出
Concurrency Diagram for 'demo'	‘Demo’并发图（多用于多线程分析）	Run to Cursor	运行到光标处
Run...	运行	Force Run to Cursor	强制运行到光标处
Debug...	调试	Resume Program	恢复程序
Attach to Local Process...	附加到本地进程	Evaluate Expression...	评估表达式
Edit Configurations...	编辑配置	Quick Evaluate Expression	快速评估表达式
Import Test Results	导入测试结果	Show Execution Point	显示执行点
Stop	停止	Toggle Line Breakpoint	切换行断点
Show Running List	显示运行列表	Toggle Temporary Line Breakpoint	切换临时行断点
Step Over	单步执行（跳过过程）	Toggle Breakpoint Enabled	切换断点启用
Force Step Over	强制跳跃执行	View Breakpoints...	查看断点
Step Into	单步执行		

16. 工具菜单

工具菜单如图 3-37 所示。

图 3-37

具体含义如表 3-9 所示。

表 3-9

原　文	中文名称	原　文	中文名称
Tasks & Contexts	任务和上下文	Create setup.py	创建 setup 文件
Save File as Template...	保存文件为模板	Show Code Coverage Data	显示代码覆盖率数据
New Scratch File...	新建临时文件	Deployment	部署
IDE Scripting Console	IDE 控制台	Test RESTful web Service	测试 RESTful Web 服务
Analyze Stacktrace...	跟踪分析堆栈	Start SSH session...	启动 SSH 会话
Capture Memory Snapshot	捕获内存快照	Vagrant	远程共享映射
Python Console...	Python 控制台	Open CProfile snapshot	打开 CProfile 快照

17. 版本控制菜单

版本控制菜单（VCS）如图 3-38 所示。

图 3-38

具体含义如表 3-10 所示。

表 3-10

原　文	中文名称	原　文	中文名称
Local History	本地历史记录	Checkout from Version Control	从版本库检出
Enable Version Control Integration...	启用版本控制集成	Import into Version Control	导入版本控制
VCS Operations Popup	弹出 VCS 菜单	Browse VCS Repository	浏览版本库
Apply Patch...	应用补丁	Sync Settings	同步设置
Apply Patch from Clipboard...	从粘贴板应用补丁		

18. 窗口菜单

窗口菜单（Window）如图 3-39 所示。

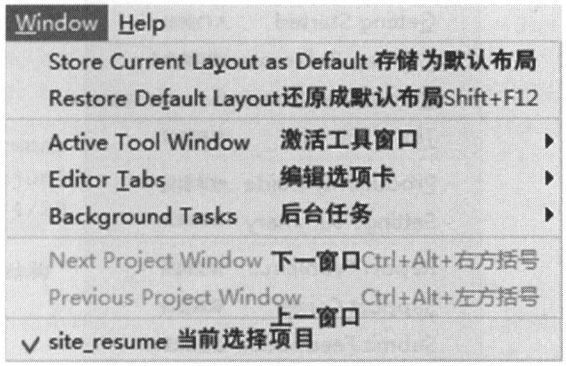


图 3-39

具体含义如表 3-11 所示。

表 3-11

原文	中文名称	原文	中文名称
Store Current Layout as Default	存储当前配置为默认布局	Background Tasks	后台任务
Restore Default Layout	还原默认布局	Next Project Window	下一个项目窗口
Active Tool Window	激活工具窗口	Previous Project Window	上一个项目窗口
Editor Tabs	编辑选项卡	site_resume	当前显示项目

19. 帮助菜单

帮助菜单（Help）如图 3-40 所示。

图 3-40

具体含义如表 3-12 所示。

表 3-12

原　文	中文名称	原　文	中文名称
Find Action...	查找动作	Support Center	联系支持
Help	帮助	Submit Feedback...	提交反馈
Getting Started	入门指南	Show Log in Explorer	打开运行日志目录
Keymap Reference	快捷键参考	Edit Custom Properties...	自定义属性
Demos and Screencasts	演示和截屏	Edit Custom VM Options...	编辑自定义虚拟机选项
Top of the Day	每日提示	Debug Log Settings...	设置调试日志
Productivity Guide	效率指南	Register...	注册
Settings Summary	设置概要	Check for Updates...	检查更新
Report Problem...	提交问题	About	关于

20. 一脚本入门 Python

```
    #!/usr/bin/env python3
# encoding: utf-8

"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/4/29

分析 SQLite db 文件结构

"""

import os  # 导入 os 模块
from sqlite3 import connect  # 导入 SQLite3 模块中的 connect 函数


class MyDbAnalysis:
    """ 自定义数据文件分析类 """

    def __init__(self, db_file):
        self.db_file = db_file  # 数据文件对对象化
        self.f = open(self.db_file, "rb")  # 以二进制方式加载数据文件,这个 db 文
#件为 SQLite3 数据文件
        self.stats = os.stat(self.db_file)  # 获取文件信息
        self.filesize = self.stats.st_size  # 文件大小字节数
        self.f.seek(0)  # 文件指针移至 0

    def db_header(self):
        """ 数据库头结构 前 100 字节 """
        head = self.f.read(100)
        print('头字符串: {}'.format(head[0:16].decode()))

        t = []  # 定义一个空的临时列表
        for i in head[16:18]:  # 使用 for 循环遍历 head 列表第 16-17 个字符
            t.append('{:02X}'.format(i))  # 使用格式化字符串将获取到的 i 转换成
#十六进制后添加到 t 列表中
        r = ''.join(t)  # 使用 join 函数将列表拼接成字符串
        pagesize = int(r, 16)  # 使用 int 函数将字符串 r 从十六进制转换成整型数字
        print('页大小: {}'.format(pagesize))  # 打印 pagesize 内容
```

```
        print('页大小: {}'.format(int(''.join(['{:02X}'.format(i) for i in
head[16:18]]), 16)))  # 此为代码 34-39 行使用推导精简

    def write_txt(self):
        """ 测试写入文本内容 """
        with open('test.txt', 'a') as f:  # 使用 with 打开文本，无须手动执行关闭
            f.write('这个是测试文本')  # 将字符串写入 test.txt 文本
        return 'OK'


def creat_db():
    """ 创建 SQLite 数据文件，并创建一个 Book 表 """
    conn = connect('record.db')  # 打开 record.db 数据库文件，如果说文件不存在，
#将会自动创建，并返回一个连接对象
    cursor = conn.cursor()  # 创建一个游标
    print(当前目录下没有 record.db 文件，所以创建 Book 数据表)
    cursor.execute("""
        CREATE TABLE Book
         (
            Title NVARCHAR(100),
            Author NVARCHAR(20),
            Price NUMERIC(8,2),
            Preface NVARCHAR(200),
            Discontinued BOOLEAN DEFAULT 1,
            PubDate DATETIME
         )""")
                ) # 执行一条 SQL 语句，这里使用了三对双引号使得 SQL 语句可以使用
    #多行文本编辑
    conn.commit()  # 提交当前的事务


if __name__ == '__main__':
    check_file = os.path.isfile('record.db')  # 检测当前目录下是否有
#'record.db'文件
    if check_file is True:
        pass
    else:
        creat_db()  # 调用 creat_db 函数创建数据文件
    try:
        d = MyDbAnalysis('record.db')  # 实例化对象
        # d = MyDbAnalysis('record1.db')  # 报错语句用于测试下面 except 块
```

```
        d.db_header()  # 调用对象中的 db_header 函数
        # d.write_txt()  # 测试 with 关键字作用
    except Exception as e:
        print('出错了：{}'.format(e))
    finally:
        a = lambda x:'不管上面哪个块执行后这里的内容都执行' + x  # lanbda 简易使
#用
        print('{}'.format(a('!')))
        # assert a == '1'  # 断言返回错误
        assert '1' == '1'  # 断言返回成功
        print('上面断言成功打印')
```

第4章
Python开发工具——Vim使用

为什么会选择 Vim 呢？Vim 的设计理念是让程序员手不离键盘地编写代码，所有操作都是通过键盘而不是通过鼠标来完成的。在 Linux 系统下，我们更多的是与控制台打交道，而 Vim 编辑器又比 vi 编辑器强，所以学会使用 Vim 对我们很有帮助。

4.1 安装 Vim

（1）在 Ubuntu 下安装 Vim

在 Ubuntu 系统下打开终端，输入下列命令完成安装，如图 4-1 所示。

```
$sudo apt-get update  # 更新软件列表
$sudo apt-get install - y vim  # 安装 Vim
```

```
xyj@xyj-OptiPlex-9020:~$ sudo apt-get update
[sudo] password for xyj:
Hit:1 http://cn.archive.ubuntu.com/ubuntu xenial InRelease
Hit:2 http://cn.archive.ubuntu.com/ubuntu xenial-updates InRelease
Hit:3 http://download.virtualbox.org/virtualbox/debian xenial InRelease
Get:4 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Hit:5 http://ppa.launchpad.net/hzwhuang/ss-qt5/ubuntu xenial InRelease
Hit:6 https://apt.dockerproject.org/repo ubuntu-xenial InRelease
Hit:7 http://archive.canonical.com/ubuntu xenial InRelease
Hit:8 https://download.docker.com/linux/ubuntu xenial InRelease
Get:9 http://security.ubuntu.com/ubuntu xenial-security/main amd64 DEP-11 Metadata [54.5 kB]
Get:10 http://security.ubuntu.com/ubuntu xenial-security/main DEP-11 64x64 Icons [50.8 kB]
Get:11 http://security.ubuntu.com/ubuntu xenial-security/universe amd64 DEP-11 Metadata [35.8 kB]
Get:12 http://security.ubuntu.com/ubuntu xenial-security/universe DEP-11 64x64 Icons [57.0 kB]
Fetched 300 kB in 19s (15.7 kB/s)
Reading package lists... Done
xyj@xyj-OptiPlex-9020:~$ sudo apt-get install -y vim
```

更新软件列表

安装Vim

图 4-1

（2）在 Windows 下安装 Vim

在 Windows 下，从 Vim 官网 http://www.vim.org/download.php 下载 Windows 版本的 Vim，

如图 4-2 所示。

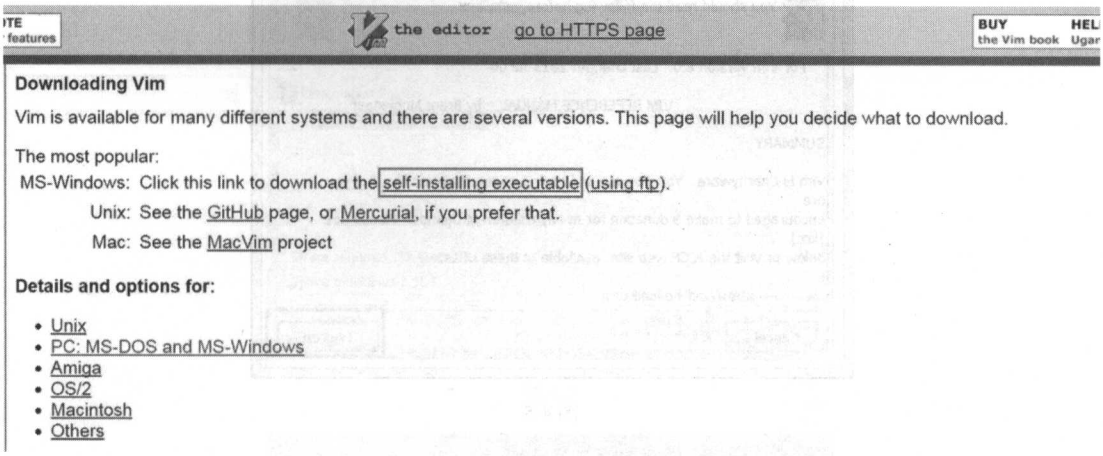


图 4-2

下载好安装包后，直接双击安装，如图 4-3 所示。

图 4-3

提示是否安装 Vim 到电脑，单击是按钮，如图 4-4 所示。

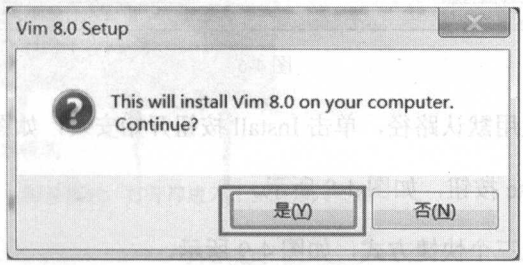


图 4-4

弹出许可协议，单击 I Agree 按钮，如图 4-5 所示。

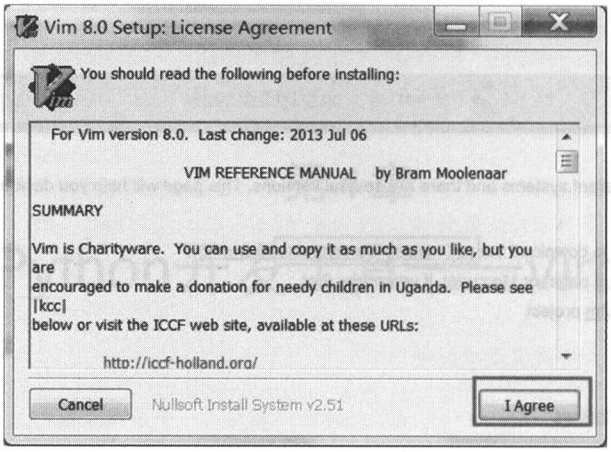


图 4-5

安装选项，默认即可，如图 4-6 所示。

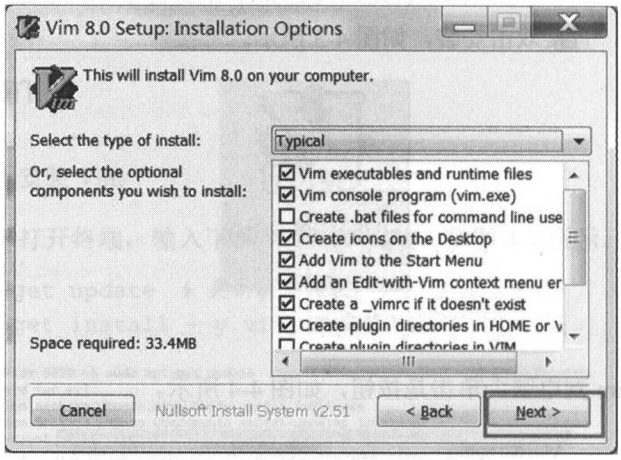


图 4-6

选择安装路径，这里用默认路径，单击 Install 按钮开始安装，如图 4-7 所示。

安装完成，单击 Close 按钮，如图 4-8 所示。

这时在桌面就会生成三个快捷方式，如图 4-9 所示。

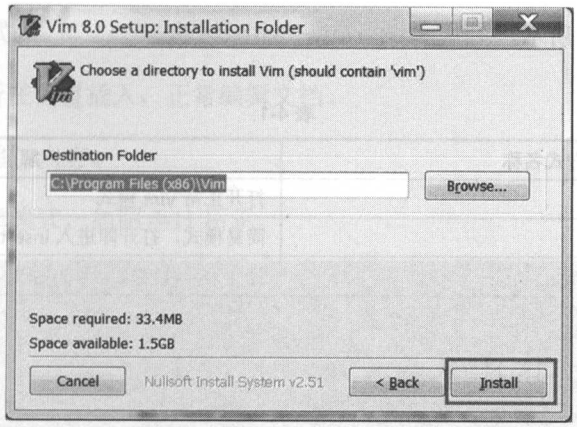

Vim 8.0 Setup: Installation Folder
Choose a directory to install Vim (should contain 'vim')
Destination Folder
C:\Program Files (x86)\Vim
Browse...
Space required: 33.4MB
Space available: 1.5GB
Cancel
Nullsoft Install System v2.51
< Back
Install

图 4-7

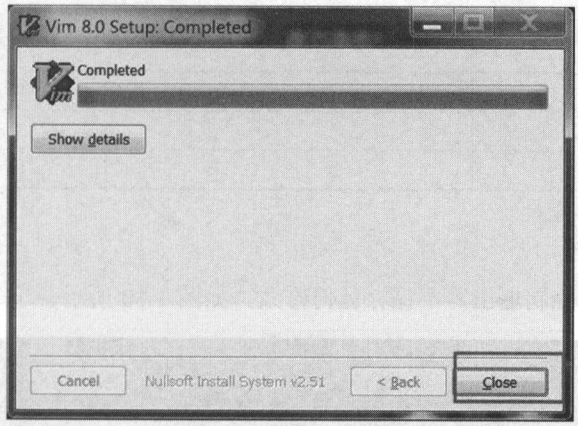

Vim 8.0 Setup: Completed
Completed
Show details
Cancel
Nullsoft Install System v2.51
< Back
Close

图 4-8

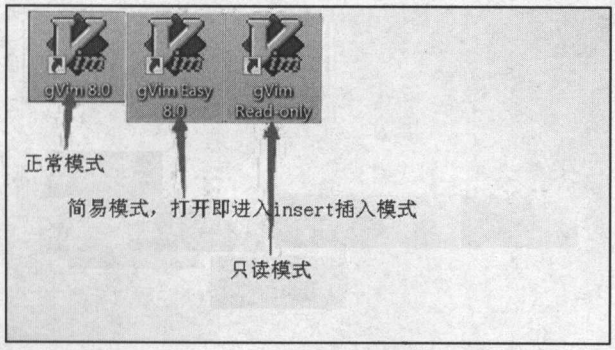

gVim 8.0
gVim Easy 8.0
gVim Read-only
正常模式
简易模式，打开即进入insert插入模式
只读模式

图 4-9

快捷方式如表 4-1 所示。

表 4-1

快捷方式名称	解　释
gVim 8.0	打开正常 Vim 模式
gVim Easy 8.0	简易模式，打开即进入 insert 插入模式
gVim Read-only	打开为只读模式，无法更改文件内容

4.2 Vim 基本使用

（1）光标移动。

键盘上 HJKL 这四个键可移动光标位置，通过 HJKL 分别控制左下上右。

	上 K	
左 H		右 L
	下 J	

（2）退出编辑器。

输入：q 回车即可强制退出，不保存任何修改，如图 4-10 所示。

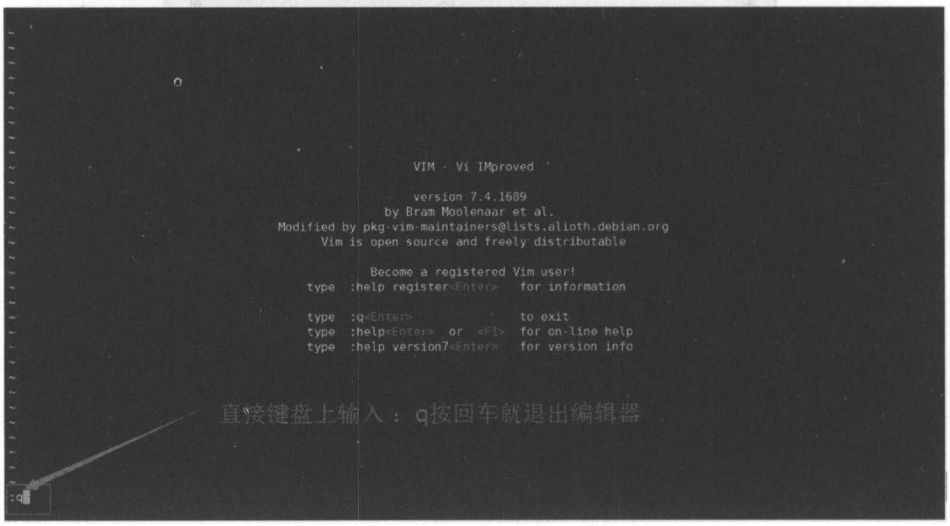


图 4-10

（3）进入编辑模式。

按 i 在当前光标所在位置插入，正常编辑文档。

按 a 则在光标后追加内容。

按 Esc 可退出编辑模式，如图 4-11 所示。

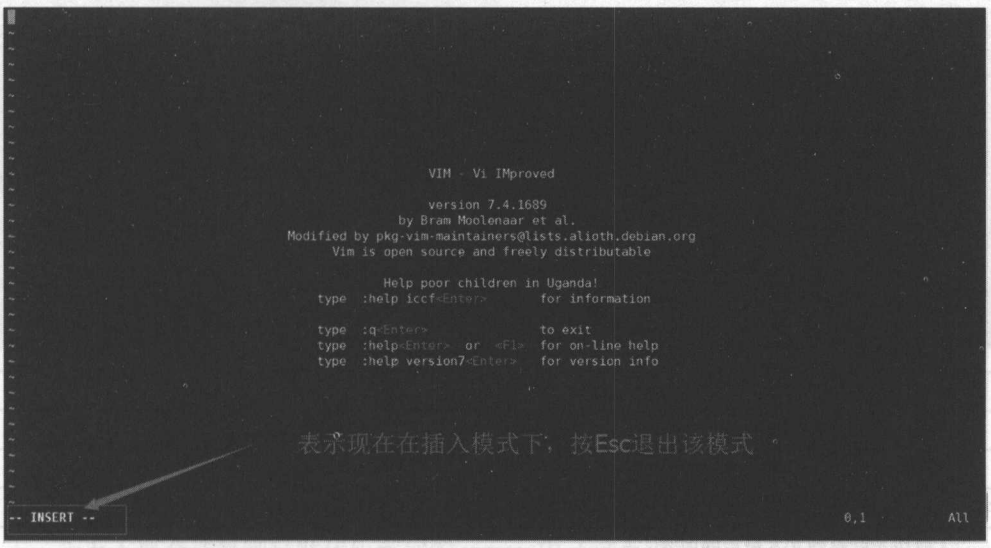


图 4-11

（4）删除内容。

非编辑模式下，按 x 可以删除光标所在位置的字符，也可以连续按两次 d 整行删除。

（5）撤销和重做。

在非编辑模式下，按 u 可以撤销之前的操作步骤，按快捷键 Ctrl+r 可以重做。

（6）复制、剪切和粘贴。

复制：在非编辑模式下，按 y（为 yank 首字母）复制当前光标所在内容，通过组合键可以实现如表 4-2 所示功能。

表 4-2

按键	实现功能，实例：Hello Wold！假如光标在 o（粗体）
y	复制当前光标所在位置内容，复制内容为 o
yy	复制光标所在整行内容，复制内容为整行 Hello Wold！
y^	复制光标前到行头的内容，按 y^后复制内容为 Hello W
y$	复制光标起到行结尾的内容，包含光标所在位置内容，复制内容为 old！
y 数字 w	复制光标起数字个内容，分割标识为符号或空格。按 y2w 复制内容 old！
yG	复制光标起到文本最后的内容

粘贴：在非编辑模式下，按 p 可将复制或剪切的内容粘贴到光标之后。

剪切：在非编辑模式下，按 d（为 delete 首字母）可剪切当前光标所在内容，组合键功能如表 4-3 所示。

表 4-3

d	剪切当前光标所在内容，剪切内容为 o，剩余内容为 Hello Wld！
dd	剪切光标所在整行内容，剪切内容为整行 Hello Wold！，剩余内容为空
d^	剪切光标前到行头的内容，按 d^后剪切内容为 Hello W，剩余内容为 old!
d$	剪切光标到行结尾的内容，包含光标所在位置内容，剪切内容为 old！，剩余内容为 Hello W
d 数字 w	剪切光标起数字个内容，分割标识为符号或空格。按 d2w 剪切内容 old！，剩余内容为 Hello W
dG	剪切光标起到文本最后的内容

（7）内容查找

在非编辑模式下，按/键或？键后输入要查找的内容，按回车开始查找内容。

/后加查找内容为，从光标所在位置往下查找匹配。

？号后加查找内容为，从光标所在位置往上查找匹配。

按 n（小写）查询下一个匹配，按 N（大写）查询上一个匹配，如图 4-12 所示。

图 4-12

3. 配置 Python 开发环境

网络上有很多介绍.vimrc 配置的文章，但要注意，在你不明白某个参数究竟有什么作用的情况下，千万不要全选复制粘贴代码到你的.vimrc 中（这点很重要）。花点时间理解每一条配置文件可有效帮助你理解 Vim 的设计理念和开放性。

（1）查看用户目录下是否有.vimrc 文件。

新安装的 Vim 默认在用户目录下是没有.vimrc 配置文件的，所以需要自己手动新建一个.vimrc 文件，如图 4-13 所示。

```
$ ls -al .vim*  # 查看用户目录下.vim 开头的所有文件详细属性，参数-a 是该目录下所
# 有文件，包含隐藏文件。参数-l 是显示当前查询显示的文件详细属性
```

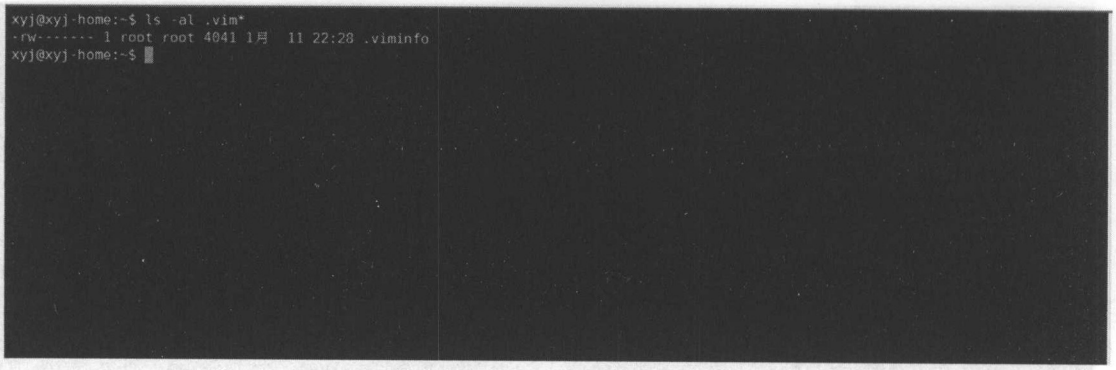


图 4-13

（2）下载 Vim 插件管理器。

Vim 插件多元化，每个插件都安装在.vim 下，当插件众多的时候，往往很难区分某个文件是哪个插件的，所以我们通过 pathogen 来进行插件管理。pathogen 在.vim/bundle 中给每个插件都分配了一个文件夹，这样通过控制这个文件夹就能使得插件的增删容易起来，如图 4-14 所示。

```
$mkdir -p ~/.vim/autoload  # 创建 autoload 目录用于自动加载插件
$mkdir -p ~/.vim/bundle  # 创建 bundle 用于保存插件目录
$curl -LSso ~/.vim/autoload/pathogen.vim https://tpo.pe/pathogen.vim
# 下载 pathogen 插件管理器
```

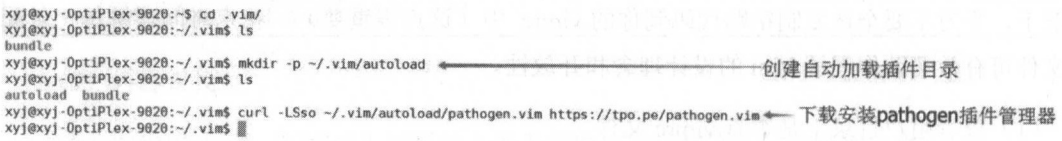


图 4-14

（3）下载 python-mode 插件。

python-mode 是一个集成了语法高亮、断点调试、PEP8 格式化、支持 Python 虚拟环境、自动 Python 缩进、代码块折叠显示、代码检查以及代码重构等功能于一体的强大插件，如图 4-15 所示。

```
$cd ~/.vim/bundle/
$git clone https://github.com/klen/python-mode.git
```

```
xyj@xyj-OptiPlex-9020:~/.vim/bundle$ cd ~/.vim/bundle/
xyj@xyj-OptiPlex-9020:~/.vim/bundle$ git clone https://github.com/klen/python-mode.git
Cloning into 'python-mode'...
remote: Counting objects: 7039, done.
remote: Compressing objects: 100% (148/148), done.
remote: Total 7039 (delta 16), reused 58 (delta 6), pack-reused 6883
Receiving objects: 100% (7039/7039), 9.74 MiB | 15.00 KiB/s, done.
Resolving deltas: 100% (2851/2851), done.
Checking connectivity... done.
xyj@xyj-OptiPlex-9020:~/.vim/bundle$
```

使用git clone
克隆python-mode插件

图 4-15

（4）创建.vimrc 文件。

新安装的 Vim 默认是没有.vimrc 文件的，需要我们手动新增。

```
$ vim .vimrc  # 用 Vim 打开.vimrc 文件，如果存在则直接打开，如果不存在则新建一个
# 文件，在当前目录下生成一个.vimrc.swp 的临时文件，如图 4-16 所示。
```

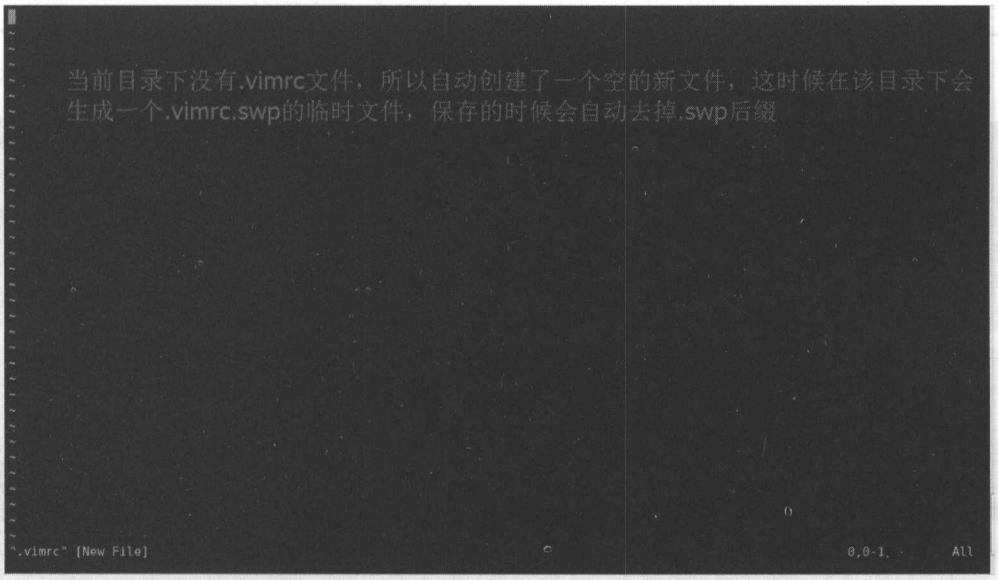


图 4-16

用第二个 shell 打开查询，得到如图 4-17 所示的临时文件。

```
xyj@xyj-home:~$ ls -al .vim*
-rw------- 1 root root  4041 1月  11 22:28 .viminfo
-rw------- 1 xyj  xyj  12288 6月   2 11:34 .vimrc.swp
xyj@xyj-home:~$
```

图 4-17

打开文件后，按 i 进入插入模式，输入下列代码：

```
"加载 vim 插件管理器
filetype off

call pathogen#infect()
call pathogen#helptags()

filetype plugin indent on
syntax on
set wildmenu  # 使用命令时按 Tab 键显示图形菜单提示，如图 4-19 所示。
```

图 4-19

（5）python-mode 的使用。

① 配置脚本运行 venv。

有两种方法指定 venv 使用的路径：第一种方法是直接在.vimrc 中指定，这样就是针对用户全局而使用的 venv（参考前面.vimrc 配置中手动指定 venv 位置的设置，把注释去掉即可）；第二种方法是进入项目后指定运行的 venv。

```
#非编辑模式下运行
:PymodeVirtualenv "/home/xyj/Pyvenv/flask-venv/"  #指定当前文件运行的虚拟
#环境路径为/home/xyj/Pyvenv/flask-venv/，如图 4-20 所示。
```

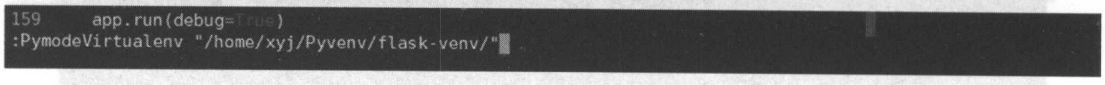


图 4-20

Python 虚拟环境激活路径，如图 4-21 所示。

图 4-21

② 快捷键。

安装好 Python-mode 插件后，可以使用表 4-4 中的快捷键实现 Python 开发中的常用功能。

表 4-4

键或命令	解　　释
\r	运行 Python 代码
\b	添加或移除断点
:PymodeLint	代码检查
:PymodeLintAuto	PEP8 格式化
K	Python 文档或帮助
:help pymode	查看 python-mode 文档
[[	转跳到上一个类或函数
]]	转跳到下一个类或函数
[M	转跳到上一个类或方法
]M	转跳到下一个类或方法

在输入模块后按.会自动提示该模块可用的属性或方法，如图 4-22 所示。

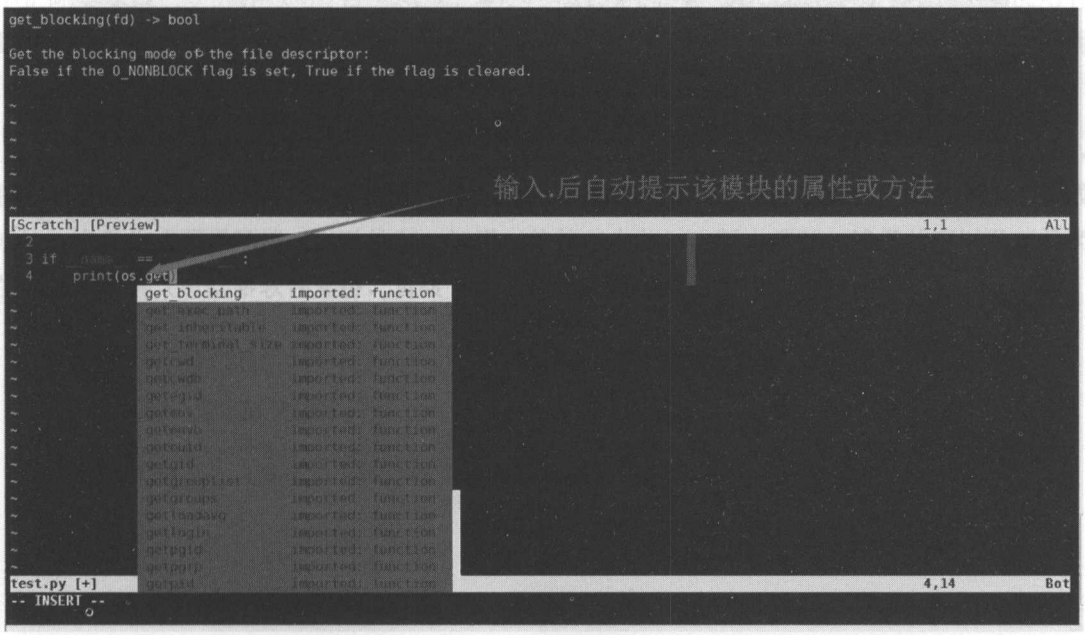


图 4-22

非编辑模式下，按\r 使用主环境 python3 解释器运行当前脚本。因为前面我们在.vimrc 中配置了 let g:pymode_python = 'python3'，指定全局 Vim 使用 python3 作为解释器，所以这里是以 python3 来解释当前脚本。或者在当前脚本中使用:PymodeVirtualenv "<path>" 来指定虚拟环境

运行脚本，如图 4-23 所示。

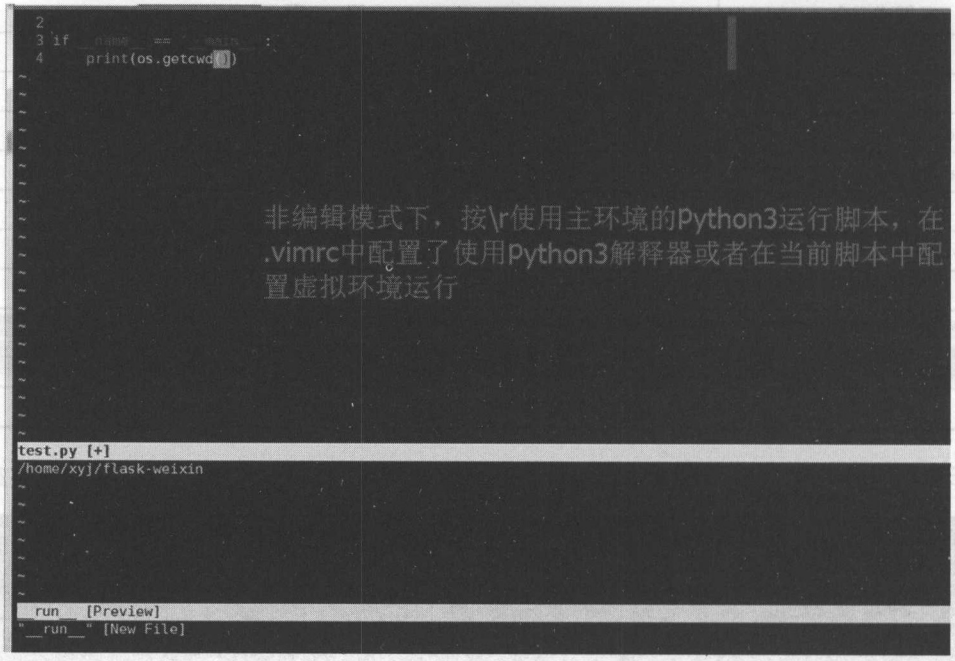


图 4-23

第5章
Docker的安装搭建

1． 为什么 Docker 火了

Docker 是什么？可能很多刚刚接触的同学会说 Docker 就是实现虚拟化的软件，这个说法不正确。Docker 只是一个应用容器化的引擎，是引擎不是实现虚拟化的软件。真正实现应用容器化的技术是基于 Linux64 位内核的 LXC 技术（这是 2013 年时的说法，在 2014 年后开始使用 libcontainer 技术，2015 年时又把 libcontainer 贡献给了 RedHat）。

原生的 LXC 技术使用起来比较麻烦，而 Docker 在这些 LXC 的基础上提供了一些高级应用，例如，跨设备的快捷部署（如果使用自定义的 LXC 部署的应用，在迁移时肯定不能立即部署成功）、自动生成预设环境（自动控制，编译，依赖包安装）、版本控制、镜像共享以及工具类生态系统完善（有众多软件支持，直接官网下载相应镜像即可使用）。

Docker 的宣传是“Build once，Run anywhere”，意思一次构建，任何地方都可运行。

2． Docker 与 VM 对比

通过对比图 5-1 和图 5-2，我们来了解虚拟机架构和容器架构的不同之处。

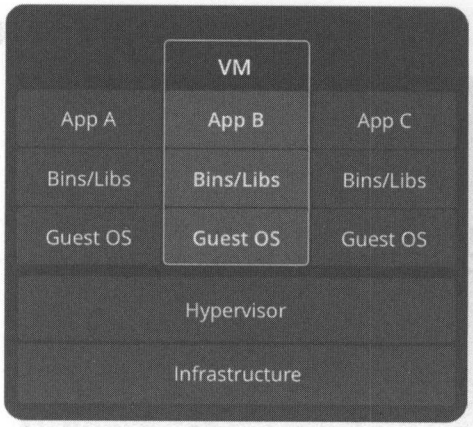


图 5-1

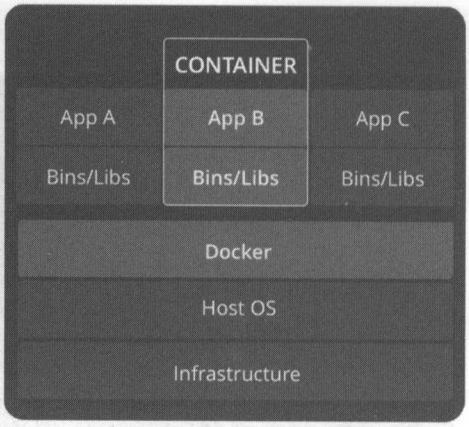


图 5-2

图来源于 Docker 官网[1]

通过对比可以看到，Docker 是建立在系统基础上的应用，通过不同的镜像实现 APP 应用的隔离，避免 APP 应用对操作系统的依赖。这样做的好处是在应用开发完成后，到部署阶段可避免系统依赖导致的应用问题。开发阶段就在镜像中运行，所有的依赖保持一致，可以无缝部署到生产环境。

3. Docker 安装

CentOS 6.5 和 Ubuntu14.04 都默认带有 Docker 软件包，但不知道为什么在 16.04 版本上面没有自带。然而 2017 年 3 月 1 日 Docker 改变了发行版本，分为 Docker-ee（企业版）和 Docker-ce（社区版）两个大版本。

企业版又细分为三个版本，包括基础版、标准版、高级版，功能对比如表 5-1 所示。

表 5-1

	社区版	企业 basic 版	企业 Standard 版	企业 Advanced 版
Container engine and built in orchestration, networking, security Docker 引擎、内置编排、网络、安全	支持	支持	支持	支持
Certified infrastructure, plugins and ISV containers 基础认证、插件、IVS 容器	-	支持	支持	支持

1 https://docs.docker.com/get-started/#containers-vs-virtual-machines

续表

	社区版	企业 basic 版	企业 Standard 版	企业 Advanced 版
Image management 镜像管理	-	-	支持	支持
Container app management 容器应用管理	-	-	支持	支持
Image security scanning 镜像安全扫描	-	-	-	支持

（1）检查内核是否支持，需要 3.10 以上内核才支持，如图 5.3 所示。

```
$ uname -r  #检查内核版本
4.4.0-47-generic  #当前内核是 4.4，高于 3.10，符合要求
```

图 5-3

（2）更新系统及软件源，启用 https 支持，安装依赖图，如图 5-4 所示。

```
$ sudo apt-get update  #更新软件源列表
$ sudo apt-get install -y apt-transport-https ca-certificates curl
software-properties-common
#安装 apt-transport-https、ca-certificates、curl 和
#software-properties-common 包
```

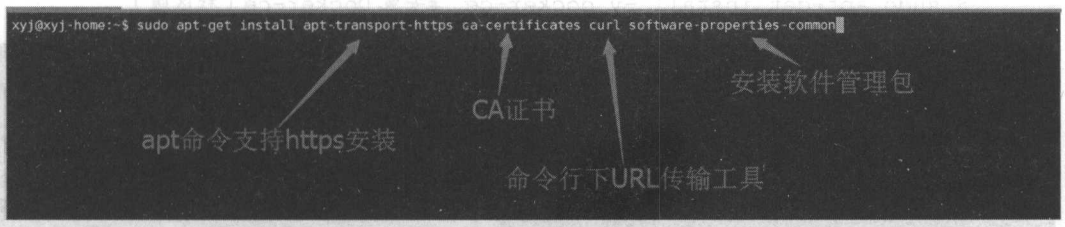


图 5-4

（3）添加 Docker 的官方 GPG 密钥，如图 5-5 所示。

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add -
```

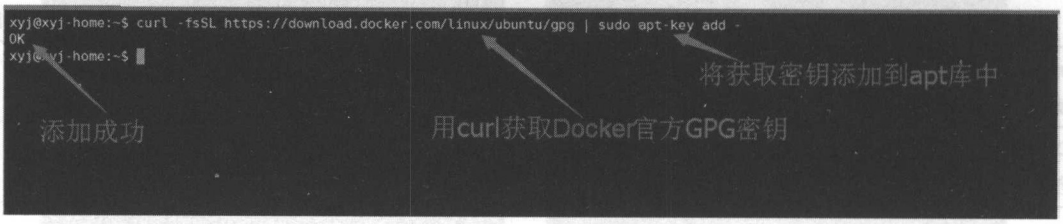


图 5-5

（4）设定 Docker 更新版本，如图 5-6 所示。

```
$ sudo add-apt-repository \
> "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
> xenial \
> stable"
```

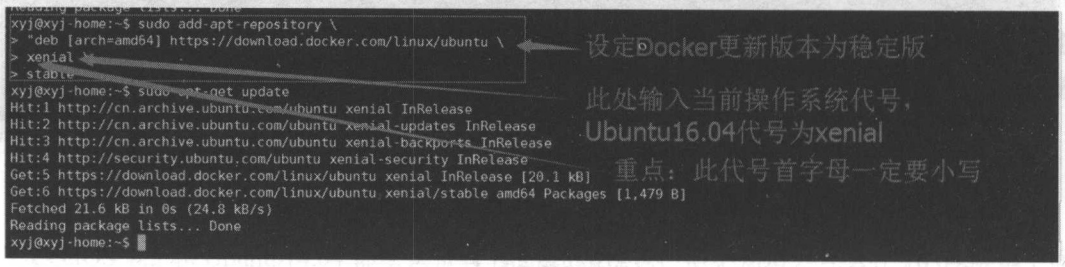


图 5-6

（5）更新软件源列表并安装 Docker-ce（社区版），如图 5-7 所示。

```
$ sudo apt-get update  #更新软件源列表
$ sudo apt-get install -y docker-ce  #安装 Docker-ce（社区版）
```

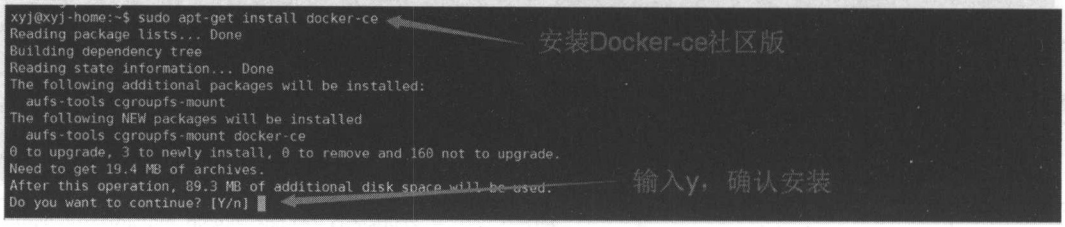


图 5-7

（6）验证 Docker 是否安装成功，如图 5-8 所示。

```
$ sudo docker run hello-world  # 用管理员权限运行 Docker
```

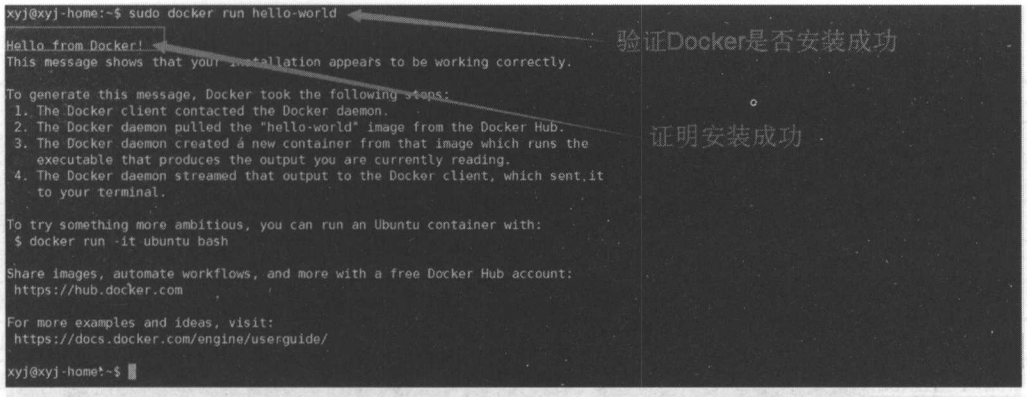


图 5-8

4. Docker 配置

前面测试安装是否正确时我们使用了超级管理员权限运行 Docker，而在生产或测试环境下，为了安全，建议使用非管理员权限来运行 Docker，因此需要配置一个单独的组来控制运行权限。

（1）创建 Docker 用户组，如图 5-9 所示。

```
$ sudo groupadd docker  # 创建 Docker 用户组
```

图 5-9

（2）将当前用户加入 Docker 用户组，如图 5-10 所示。

将当前用户添加到 Docker 用户组后，不使用 sudo 超级权限也可以控制 Docker。

```
$ sudo usermod -aG docker xyj  # 添加 xyj 用户到 Docker 用户组
$ sudo reboot  # 重启服务器
```

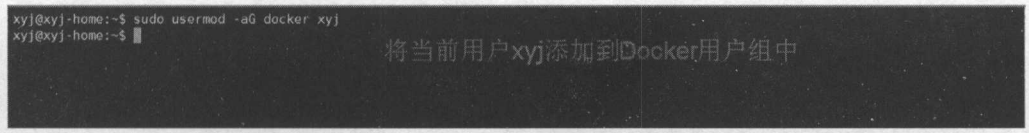


图 5-10

（3）查看 Docker 后台服务运行状态，如图 5-11 所示。

```
$ sudo systemctl status docker.service  # 查看 Docker 后台服务运行状态
```

图 5-11

注意：在低版本系统时我们常常使用 service 或 chkconfig 命令来配置服务的启动和关闭，在 Ubuntu 15.04 后开始使用 systemctl 命令来管理系统服务，代替原来两个命令去实现相应功能。

系统 Systemd 组件如图 5-12 所示。[1]

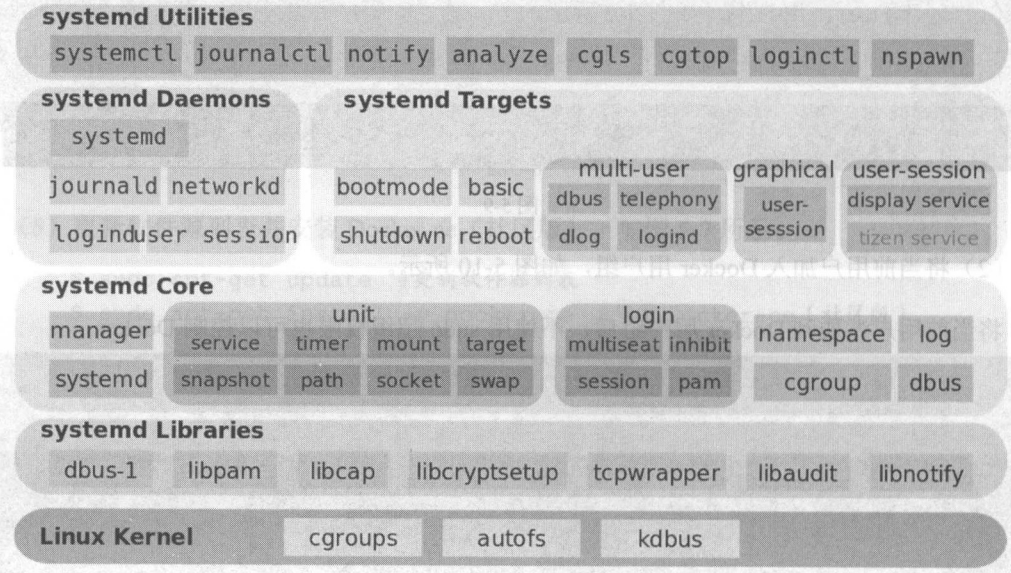


图 5-12

1 https://en.wikipedia.org/wiki/Systemd

配置服务新旧指令对比如表 5-2 所示。

表 5-2

任　务	旧指令	新指令
服务自动启动	chkconfig --level 3 <Service> on	systemctl enable <Service>.service
服务禁止自动启动	chkconfig --level 3 <Service> off	systemctl disable <Service>.service
查看服务状态	service <Service> status	systemctl status <Service>.service
显示所有已启动服务	chkconfig --list	systemctl list-units --type=service
启动服务	service <Service> start	systemctl start <Service>.service
停止服务	service <Service> stop	systemctl stop <Service>.service
重启服务	service <Service> restart	systemctl restart <Service>.service

（4）配置 Docker 自动启动。

配置 Docker 自动启动，需要禁用的时候，只需将 enable 改成 disable，执行一次命令即可禁用 Docker 自动开机启动，如图 5-13 所示。

```
$ sudo systemctl enable docker.service  # 设置 Docker 自动启动服务
```

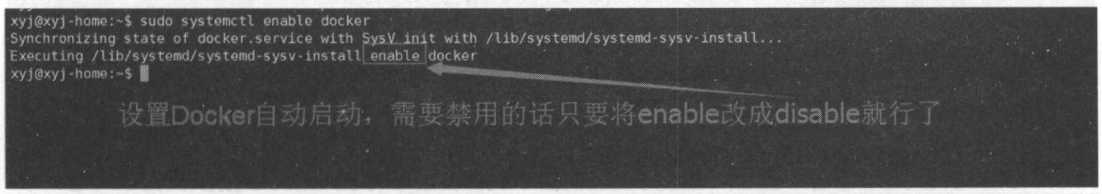


图 5-13

（5）验证 Docker 服务是否能访问网络

使用 pull 命令可以测试出 Docker 服务是否可以访问外网，如图 5-14 所示。

```
$ docker pull hello-world  # 拉取 hello-world 镜像
```

图 5-14

（6）检测 Docker 容器是否能解释内部主机名。

在终端中与容器交互使用 ping 命令检查容器是否能解释访问网络，如图 5-15 所示。

```
$ docker run --rm -it alpine ping -c4 www.baidu.com  # 运行 alpine 镜像，
#执行 ping 命令
```

图 5-15

（7）指定 Docker 的 DNS。

如果不能 ping 通，则需要给 Docker 增加 DNS 的指向 IP。

```
$ sudo vim /etc/docker/daemon.json  # 编辑或新增 daemon.json 文件
```

新增内容：

```
{
    "dns": ["10.225.30.181", "10.225.30.223"]
}
```

Docker 的 DNS 配置文件默认保存在/etc/Docker/daemon.json 下，默认安装时这个文件一般不配置，所以需要手动创建 daemon.json 文件，如图 5-16 所示。

图 5-16

添加 DNS 信息如图 5-17 所示。

图 5-17

（8）升级 Docker-ce。

使用下面命令检查是否有新版本的 Docker-ce，有则进行升级，如图 5-18 所示。

```
$ sudo apt-get upgrade docker-ce  # 检查更新 Docker-ce
```

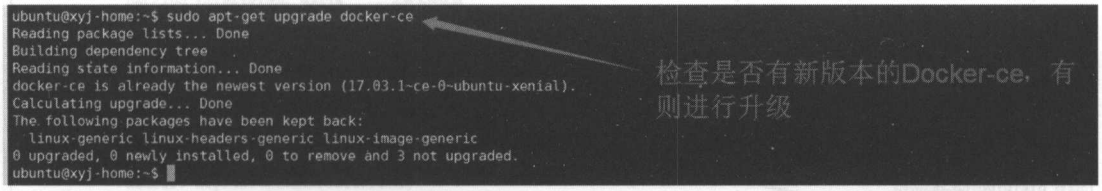


图 5-18

（9）卸载 Docker-ce。

```
$ sudo apt-get purge docker-ce  # 卸载 docker-ce，如图 5-19 所示。
```

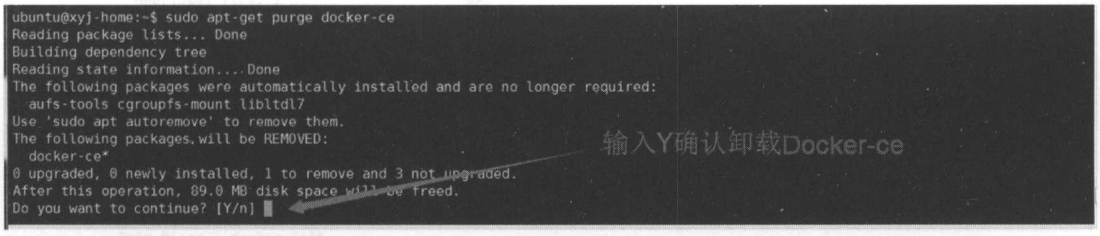


图 5-19

这个卸载只是把 Docker 的引擎给卸载了，主机上的镜像、配置、容器以及卷这些文件都没有删除，需要手动删除则执行下面命令，如图 5-20 所示。

```
$ sudo rm -rf /var/lib/docker  # 删除 Docker 的目录
```

图 5-20

（10）调整内存和 SWAP，编辑 grub 文件。

为避免出现下面的警告，需要配置调整内核参数，如图 5-21 所示。

WARNING: Your kernel does not support cgroup swap limit. WARNING: Yourkernel does not support swap limit capabilities. Limitation discarded.

```
$ sudo vim /etc/default/grub  # 编辑 grub 文件
#GRUB_CMDLINE_LINUX_DEFAULT=""  # 将这一行注释
GRUB_CMDLINE_LINUX_DEFAULT="cgroup_enable=memory swapaccount=1"
# 新增一行
```

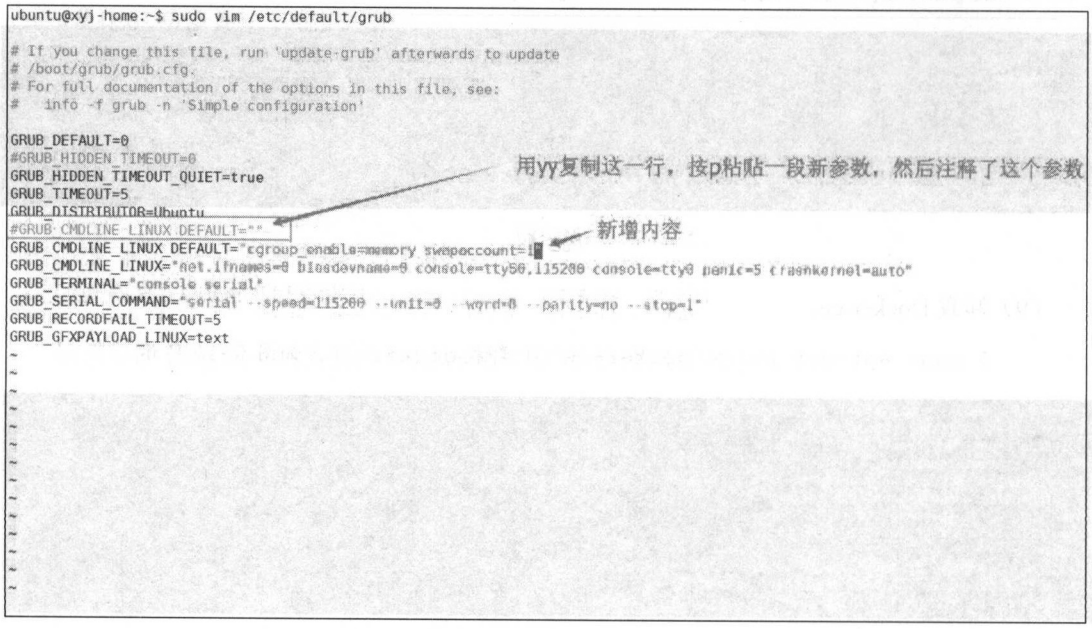


图 5-21

5. Docker 命令集

（1）查看 Docker 信息。

用 Docker info 这条命令检查 Docker 是否安装正确，如图 5-22 所示。

```
$ docker info  # 查看 Docker 信息
```

```
xyj@xyj-home:~$ docker info
Containers: 14    一共有多少个容器
 Running: 0       运行容器
 Paused: 0        暂停容器
 Stopped: 14      停止容器
Images: 51        一共有多少个镜像
Server Version: 17.06.0-ce        docker服务器版本
Storage Driver: aufs              存储驱动程序
 Root Dir: /var/lib/docker/aufs   存储根目录
 Backing Filesystem: extfs        支持文件系统
 Dirs: 143                        目录
 Dirperm1 Supported: true         Dirperm1支持
Logging Driver: json-file         日志驱动程序
Cgroup Driver: cgroupfs           Cgroup 驱动
Plugins:                          插件
 Volume: local                    存储位置:本地
 Network: bridge host macvlan null overlay    网络
 Log: awslogs fluentd gcplogs gelf journald json-file logentries splunk syslog  日志
Swarm: active                     集群:激活
 NodeID: kcy74uv4mt4q0h1z89uwzbc68
 Is Manager: true                 主控制
 ClusterID: jrb4o1wm2h1vebux09fe9po41
 Managers: 1                      主控制台数
 Nodes: 1                         节点数量
 Orchestration:                   编排
  Task History Retention Limit: 5  任务历史保留限制
 Raft:                             Raft配置
  Snapshot Interval: 10000                快照间隔时间
  Number of Old Snapshots to Retain: 0    快照保留数量
  Heartbeat Tick: 1                       心跳
  Election Tick: 3                        最低选举数
 Dispatcher:
  Heartbeat Period: 5 seconds             心跳频率:5秒
 CA Configuration:                 CA证书配置
  Expiry Duration: 3 months               证书有效期:3个月
  Force Rotate: 0                         强制替换
 Root Rotation In Progress: false
 Node Address: 192.168.1.137              节点IP
 Manager Addresses:                       管理节点IP
  192.168.1.137:2377
Runtimes: runc                            执行引擎
Default Runtime: runc                     默认执行引擎
Init Binary: docker-init
containerd version: cfb82a876ecc11b5ca0977d1733adbe58599088a    容器版本
runc version: 2d41c047c83e09a6d61d464906feb2a2f3c52aa4          引擎版本
init version: 949e6fa                                           初始化版本
Security Options:          安全选项
 apparmor                         AppArmor安全策略
 seccomp                          安全计算模式
  Profile: default
Kernel Version: 4.4.0-91-generic        系统内核版本
Operating System: Ubuntu 16.04.3 LTS    操作系统
OSType: linux                           系统类型
Architecture: x86_64                    CPU架构
CPUs: 4                                 CPU核数
Total Memory: 7.717GiB                  物理内存
Name: xyj-home                          主机名
ID: ZGC2:QEY4:6Y0A:3BUE:QX7H:7JC6:LVWA:2GZE:FQ6Z:25W0:5TIJ:A4A0
Docker Root Dir: /var/lib/docker        docker根目录
Debug Mode (client): false              客户端调试模式
Debug Mode (server): false              服务器调试模式
Registry: https://index.docker.io/v1/   镜像储存位置
Experimental: false                     实验
Insecure Registries:                    不安全仓库
 127.0.0.0/8
Registry Mirrors:                       版本仓库
 https://ccwdulta.mirror.aliyuncs.com/
Live Restore Enabled: false             实时还原

WARNING: No swap limit support          警告信息
xyj@xyj-home:~$
```

图 5-22

Docker 信息翻译如表 5-3 所示。

表 5-3

Containers: 1	一共有多少个容器
Running: 0	运行容器
Paused: 0	暂停容器
Stopped: 1	停止容器
Images: 2	一共有多少个镜像
Server Version: 17.03.1-ce	Docker 版本
Storage Driver: aufs	存储驱动程序
Root Dir: /var/lib/Docker/aufs	根目录
Backing Filesystem: extfs	支持文件系统
Dirs: 4	目录
Dirperm1 Supported: true	Dirperm1
Logging Driver: json-file	日志驱动程序
Cgroup Driver: cgroupfs	
Plugins:	插件
Volume: local	卷
Network: bridge host macvlan null overlay	网络
Swarm: inactive	集群
Runtimes: runc	执行引擎
Default Runtime: runc	默认执行引擎
Init Binary: Docker-init	
containerd version: 4ab9917febca54791c5f071a9d1f404867857fcc	容器版本
runc version: 54296cf40ad8143b62dbcaa1d90e520a2136ddfe	引擎版本
init version: 949e6fa	初始化版本
Security Options:	安全选项
apparmor	AppArmor 安全策略
seccomp	安全计算模式
Profile: default	
Kernel Version: 4.4.0-53-generic	系统内核版本
Operating System: Ubuntu 16.04.2 LTS	操作系统
OSType: linux	系统类型
Architecture: x86_64	CPU 架构
CPUs: 1	CPU 个数

续表

Total Memory: 864.5 MiB	物理内存
Name: xyj-home	主机名
ID: ISIB:NZEB:BKJL:XMEY:Y65J:WYTR:4L57:VRYY:WPSU:AP7K:B53T:I7WL	
Docker Root Dir: /var/lib/Docker	Docker 根目录
Debug Mode (client): false	客户端调试模式
Debug Mode (server): false	服务器调试模式
Registry: https://index.Docker.io/v1/	镜像储存位置
WARNING: No swap limit support	警告信息
Experimental: false	实验
Insecure Registries:	不安全仓库
127.0.0.0/8	
Live Restore Enabled: false	实时还原

这里有一条警告信息，关于交换分区（swap）不支持限制功能（可以通过前面 Docker 配置的第 10 点方法解决）。

（2）切换成国内源。

众所周知，我们访问 Docker hub 时存在部分困难，所以这里切换成国内的镜像源，以便在拉取官方镜像时会有较快的速度。可以通过修改 daemon 配置文件/etc/Docker/daemon.json，添加 Docker-cn 提供的 Docker 镜像源加速，如图 5-23 所示。

```
    $ sudo vim /etc/docker/daemon.json  # 修改/etc/default/docker
    "registry-mirrors": ["https://registry.docker-cn.com"]  # 新增使用
#Docker 镜像加速
    $ sudo systemctl daemon-reload  # 重新加载配置
    $ sudo systemctl restart docker.service  # 重启 Docker 服务
```

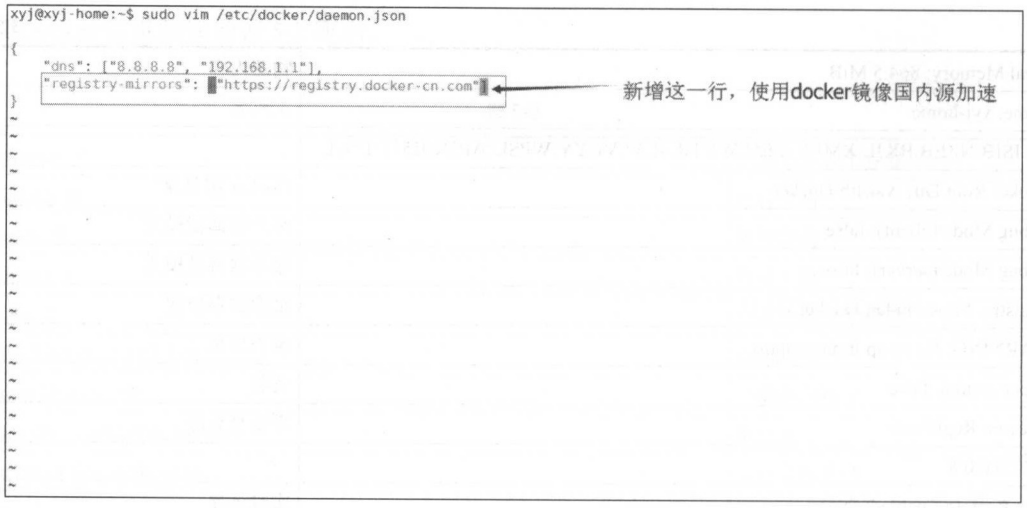


图 5-23

（3）查看 Docker 版本信息。

```
$ docker version  # 查看 Docker 版本信息，如图 5-24 所示。
```

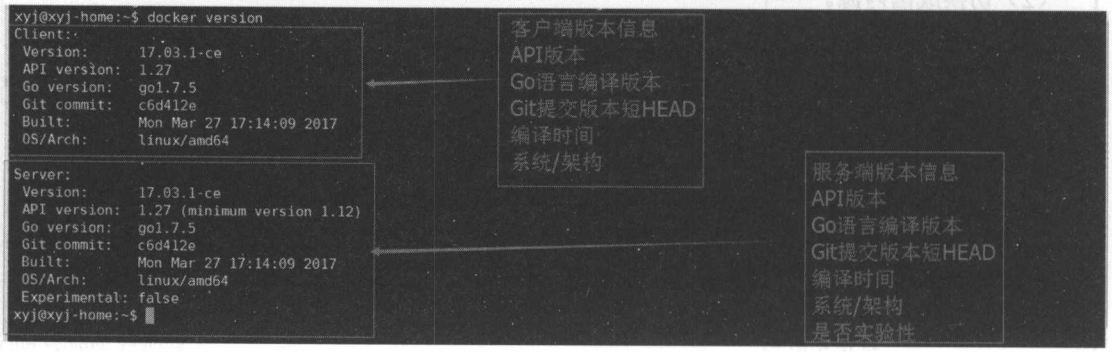


图 5-24

（4）拖取镜像。

在控制台用 Docker 命令通过 pull 参数从 Docker hub 服务器上拖取别人创建好的镜像，如图 5-25 所示。拉取的镜像一般会存放在 Docker Root Dir\<Storage Driver>下的 layers 文件夹中。Ubuntu 默认使用 aufs 驱动，所以默认镜像的保存位置是/var/lib/Docker/aufs/layers，如图 5-26 所示。

```
$ docker pull busybox  # 从 Docker Hub 上获取 busybox 镜像
```

```
xyj@xyj-home:~$ docker pull busybox  通过docker pull命令 从docker hub上获取busybox镜像
Using default tag: latest
latest: Pulling from library/busybox
add3ddb21ede: Pull complete
Digest: sha256:b82b5740006c1ab823596d2c07f081084ecdb32fd258072707b99f52a3cb8692
Status: Downloaded newer image for busybox:latest
xyj@xyj-home:~$
```

图 5-25

```
xyj@xyj-home:~$ sudo ls /var/lib/docker/aufs/layers   通过pull拉取的镜像保存在Docker Root Dir下
[sudo] password for xyj:
012090b9887d8f62c531bc11f6b2d0a63950e7b5550b77ebee47aa65cb9d8d39
05a24b1fec6c7093068f6bdb34ddebf2e12bf9d12feebc943527b3c7f11361af
071ee1f128ae269f687b27a294a854a7797778d1faf9dc0cf9f25bba6f7884e69
0729cfb5d0e474bd0d350ecb209abf1e4db616cc98f2e563a31b74e70b4b37e5
07675386b1dd8ab716c584a684d8c2e17d84138ca693825bc053a8dceac7b066
0fa619798a7d651e37bb0d090b3d917b8446c9b3411f132c6700fc8200c9369c
0ff6fce497dafc871c1b3ae2f4021dd2d7c13d5ba27f8df74ffc06ce837b585b
139348f169736120b3d141d73b1223d0d84eb50bdb5f4a72f682f7b89f3d3a6d8
13f84892fd230b2c1f205c7cc6477d90c022d89c0a4b3d82f3cf39dc0d960ede
14d485cb0cfbd70678f251dabfeefe604b002b04a6a9543c4e8b1536a084f1ca
154459b0b3ba808d0fa268e31de1113521e928b7c1124716c6367fd752b18841
17db0384d6457e25769cefb50504d8d9b734a190c763b8b68472de37124ad5ec
1a24c80bda0b330498a906e24042722a639889f953c072a9c40fb4bccdaa39748
1a24c80bda0b330498a906e24042722a639889f953c072a9c40fb4bccdaa39748-init
1b6bae7a78adb20acf438baf98e84b7c88b578aa85f932b08044c370268ab903
1cb9dee911160063ba7b50e9be5e314af8c105074dfa1c3317cd73e834f8a08bd
1d6837aaf952ec27fd14c0c0707c8fc3c20da778aaebc3c7907f8c54285e5739
22983ba939b8c752ed6487f1f28cba178ca6ac0776180fc2ad89de6b4a905247
2647636416c9eb06c81b667e1cfd8d6cd35f8a2b855086d4cfc1461afc6ac86c
26f13ee5c903f1c567df98f868d117ca433ef3489b7bc893f9d1c2c0f6120f19
27fce634359810da74d734146ce9d7a699c3a97326a08dd8ae5373afbcd32a4eb
2aef2e46c246ef4728b33bb28892a3ee7bea09bf3a55dd399ebc0c2c4060b5eb
2b229dc6034e6d8d2410e6fe08a772e88b7a2a3c21dee5817d2a05bda9ec6e46
2d711edb3c8a9034652c2e4dae79d399dc0c4bb82de156aebd65bb419e6c6348
2e72bc83ec9b64f4ab61afe2cfdee5ac425810f2622004cd9a4f81798ab67f26
32bf753fe7e824d3dcea247deb828bc2a0927777b3b99950f77ea0fd88b6cfcf
34c7de6cd23900e9c5b85120be9eb65ef934f57fa5370767d2ff3b8a4a797ff6
36199f329f005ebfd81303d9fd1663a77904753o6ff23d2a1ef6632c7b773227
36199f329f005ebfd81303d9fd1663a77904753o6ff23d2a1ef6632c7b773227-init
379c856e50d2455ffa5f7661e1f411b1e602f3169661471f61b0ac2b6714e2a2
388c5609089729b47edf012d26aae6b46b798f7ac1abc9ab99e24a60169c76ed
38b4ab7ab1f3298f5724271958b6d9ffc55039feea5b72a0f82180c91ad8fdd2
3add76c47065ea77e1d13772f3fb099202d4185c3db99bf671fa9a416ba88a91
3afc9f322ee7a628add91db9232d3af1a915f4a97561b0ba55f422fc00f5a020
3bdbc49fcd691a956666dad91366a0009c83cdac362c0fd8a7579b2cea71524d8
405891b29b29calc2760914fbbcbbb33c1f9d2a54e65b5c368940adaaad0baae
42f053f869bbf8e5d50dc2a5d029073516d2b4cda2a0e0c9239d23226acf5479
42f053f869bbf8e5d50dc2a5d029073516d2b4cda2a0e0c9239d23226acf5479-init
43b55a51efa140775d3730d46240b8e0660efe3677a13c2ea7e7226925e9da7f
480f82af4de5f58588e55cd56af781cfc026afceba8910ad05f90da7bcdd9f47
49ab656c0fe280956719e6fdb42431d7143567e9206106667b120c2f95ee2248
4b4cb64d508ed95499c917b25b2892cfa7070956c1bde6d6683c4702ddc89a89
4c252e09726fdb8b987c23dbef6c904739e0af1452029ec07b4c6110ab25b8f3
7963272d896eb9c3c9ed2591e60973181f666d1ac360f87c2c3ab18be8bc90af
7c36739a48b38cca8a0a76810e3c23fba4b59373f75e595f609f2c7154ec4f93
7dbe0c62ee56fed55478540c75d6eba329e9dbd98dcc7240d9930a18e9868144
7f97f45e3d1e674a303bd30865cedf49e56f47b304df12a7da60d92133b337a1
835bc87adad3191d5e07a2d240b376aaaf4dd714526968866d7c76294486e0994
845e8698c74f2a68ec6e6e7cde9b17da775efefbf7513f0170dfdc4c41b985dd
845e8698c74f2a68ec6e6e7cde9b17da775efefbf7513f0170dfdc4c41b985dd-init
88f05e7f4b9d064fab370a2a5d9337c7a49c5e03db9331d55af48d5ec4a40570
8bbc66421abff545311f9dd561be77df906a7ea752eada5094af2a3bf2734ba5
8d6054884600081 1a57db87627772a3485118724bea6cb0813b6f684fce57822
8f439d21c6dba60d0fb0162586778e33988c9b8c1ee033623cc5674a38f6f6b1
928c150aa1865732ba7d08195d80a6f77ae8f775f02246e3b519b0af67f84275
92b025884fe1a4de14283da6738d71e75e90092493d493a953512c07466479fd
9490fc2839dc5afb41d02709fbca98b3c8791105 18ba1245c32f945e9faf2e75
9490fc2839dc5afb41d02709fbca98b3c8791105 18ba1245c32f945e9faf2e75-init
9590e1f1c67032cef7838ef7fc486c32cb20c29fae3b28deb2b887b7793bc5cf
95ad91b9fcec66bf98c167a6e30cbe188e1a4cb855585f47c9c52258e0b15b386
98bb86d902f80edbc034de366bbdd59ea64f5539930c21efb58b5aa64ab8ae44
98c5df4bc2016689ad8f957dd18c043f3be4ccbf7b48edc3e6ef34b76674fbc3
99ba54eda33fdf91f1029ba5c697f193b246b3853937796e779ae534b228a135
9a9b3a56c65316fde594d9a623477ea7ab2757631c23955e43461e1275d05578
9b79743ad44164b4e45d459117b0b349cb5e5534ea58fc11a2f6c167ce2ede12
9cb5b4ff92d2445c48f1daf221b3db893e70fcb3eadd31bc5aa7e93260c4efd7
9ce7593716655253478629a7c7196aa114acf1dc2cd211615f38d5ec5f129832
9dcc5ad85e2da341c13b2054a1a79c61a59c245e4c76c7a905bee2607c0c0d17
a0285ffd19e96eaea43e083809008dd98979a6dbbe3b54be3172dc7a3e52abc1
a0b78ea2793019d658559a4834ace1f23918f7123d02b4ed70ac9b030c9248d6
a1329efbf6fe26a65ae8c7d30e2fb01d431328d935f3470749382 60eb2a1da74
a3151509eb132dcdc705e8fa9cc5a939fed7b849a50ea24ef383cd477c79c13a
a435262631f839a788dd438e65ca4e96dd215536a67522aa67c3fa8440f7a452
aac13f6a13ef008861350424d2dd4e6e049ba3fd69dc17d1936f1348abd30740
abf042a162f585b226892eafbaca016c9b0d56e5d265a44422e856d5e2cae548
ac1d3d8a2e0a5292c0d6c5160ed66e837d0d6880666105439e9d0c48582f27f1
ac1d3d8a2e0a5292c0d6c5160ed66e837d0d6880666105439e9d0c48582f27f1-init
ac5cbb9b422e4100829617689b1fb2a5baa9183d67842278 65c0c66bc2c4e841
acecf2bdad4be3f8e809e0b0027b7ea86d144d98f35a8d0936e6a4204381f92b
b0fb2e3fa69f363c2842575ef5161aa23c9f43cf247388e228431fd4266c7b1b
b3860fdcf6459b9b865ac3bb372cc781199f8e9487ce3f9fa3c57a630005c682
b4b14f7e70d29eefb143f9c3c80b398b40721cc1d180cba3449e5865e42f3ac7
b72dbd2ec92c834f85ef6c8dbd5d2160cc5bb78b7ca0c1a0dad9dd2b37fc8dbb
b8b427665fb7faca0a221f4a9fafe421db4b2e265f66cfc257a2b911062ea56a
bcdbcebb7f735bfc6ab0c92229e779453864e3bccfe9b9e7d8f4d2e019fd2c1c
bee7ecd7e1912ef2eb51121ac9e65ed3dc44360ac5445575bb40f9bbacf48451
```

图 5-26

注意：Docker 官方提供了一个类似 GitHub 的空间，用于镜像或资源的托管共享。上面有很多已经构建完成的镜像，通过 pull 拉取下来就能直接使用。官方网站是 https://hub.Docker.com/，最近官方宣布 Docker Hub 即将被 Docker Store[1]和 Docker Cloud[2]取代。

Storage Driver（存储驱动程序）的主要分类及 Linux 系统支持的驱动如表 5-4 所示。

1　https://store.docker.com/

2　https://cloud.docker.com/

表 5-4

支持 Linux 系统	存储驱动程序				
	aufs	devicemapper	overlay Linux 系统支持的驱动（数据来源于 Docker 官网）[1]	overlay2	zfs
Ubuntu	默认	支持	支持	-	支持
Ubuntu14.04.4 及以上版本	默认	支持	支持	支持	支持
Debian	支持	支持	支持	支持	-
CentOS	-	支持	-	-	-
Fedora	-	支持	实验支持	实验支持	-

各个驱动对比如表 5-5 所示。

表 5-5[2]

存储驱动	存储数据级别	稳定性
aufs	file-level（文件级）	稳定
devicemapper	block-level（块级）	稳定
overlay	file-level（文件级）	稳定
overlay2	file-level（文件级）	较新
zfs	block-level（块级）	较新

（5）简单示例。

在 Docker 命令后使用 run 参数运行 busybox 镜像，后来的内容是在镜像中运行的命令，格式如下：

```
docker run <imgae> [image_shell]
$ docker run busybox /bin/echo Hello World  # 运行 busybox 镜像，在容器内执行
#命令，如图 5-27 所示
```

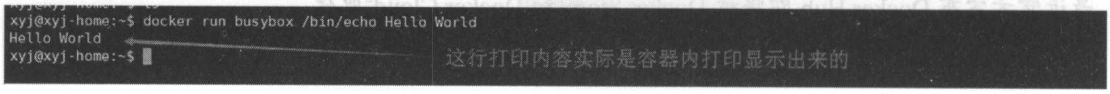


图 5-27

1 https://docs.docker.com/engine/userguide/storagedriver/selectadriver/#docker-ee-and-cs-engine

2 https://docs.docker.com/engine/userguide/storagedriver/selectadriver/#supported-backing-filesystems

（6）查看容器。

```
$ docker ps  # 查看当前在运行的容器
$ docker ps -a  # 查看所有容器信息，如图 5-28 所示
```

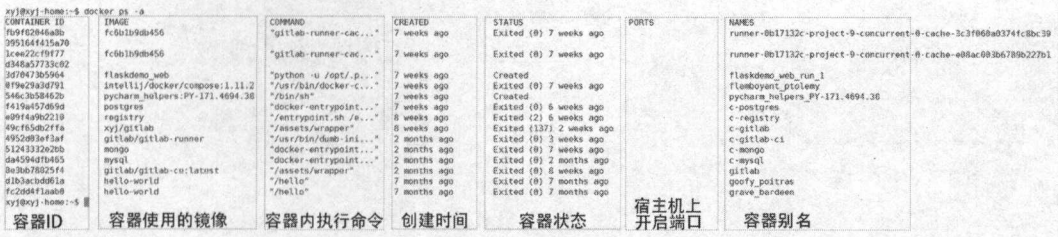


图 5-28

（7）与容器交互。

```
$ docker run -t -i busybox  # 运行 busybox 镜像，进入 tty（仿真终端）并开启交互，
#如图 5-29 所示
```

```
xyj@xyj-home:~$ docker run -t -i busybox    运行busybox镜像，进入tty（仿真终端）并开启交互
/ # ls    容器内执行ls命令
bin   dev   etc   home  proc  root  sys   tmp   usr   var
/ # exit  退出容器
xyj@xyj-home:~$
```

图 5-29

（8）停止容器。

为了演示容器正在运行的状态，笔者在后台使用了另外一条线程与 busybox 镜像进行交互，所以通过 ps 可以查看到当前有一个容器正在运行。可以使用 Docker stop <container_id>结束正在运行的容器，如图 5-30 所示。

```
$ docker stop bc9237c2cf64  # 停止 ID 为 bc9237c2cf64 的容器
```

```
xyj@xyj-home:~$
xyj@xyj-home:~$ docker ps 通过docker ps命令查看容器状态，发现有个容器在后台运行
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
b9c54934f02e        busybox             "sh"                22 seconds ago      Up 21 seconds                           musing_murdock
xyj@xyj-home:~$ docker stop b9c54934f02e  使用docker stop <container_id> 停止指定容器
b9c54934f02e
xyj@xyj-home:~$ docker ps 已结束id为b9c54934f02e的容器
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
xyj@xyj-home:~$
```

图 5-30

（9）重新启动容器。

容器运行结束后，可以通过 restart 命令使得容器重新运行，运行方式可以通过容器 ID 或容

器别名执行，如图 5-31 所示。

```
$ docker restart 3d544c85e2c8  # 使用容器 ID
$ docker restart hopeful_lalande  # 使用容器别名
```

```
xyj@xyj-home:~$ docker ps -a                    给docker ps增加参数-a，查看全部容器信息
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
b9c54934f02e        busybox             "sh"                28 minutes ago      Exited (137) 27 minutes ago                     musing_murdock
57e348494857        busybox             "sh"                28 minutes ago      Exited (0) 28 minutes ago                       clever_almeida
xyj@xyj-home:~$ docker restart b9c54934f02e   使用容器ID重启容器
b9c54934f02e
xyj@xyj-home:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
b9c54934f02e        busybox             "sh"                37 minutes ago      Up 2 seconds                            musing_murdock
xyj@xyj-home:~$ docker stop b9c54934f02e      停止该容器
b9c54934f02e
xyj@xyj-home:~$ docker restart musing_murdock 使用容器别名重启容器
musing_murdock
xyj@xyj-home:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
b9c54934f02e        busybox             "sh"                37 minutes ago      Up 2 seconds                            musing_murdock
xyj@xyj-home:~$
```

图 5-31

（10）移除容器。

创建的容器运行完成后还会存在，每运行一个新的容器就会创建一个新的层，所以当我们能够确定不使用某个容器时可以手动移除，如图 5-32 所示。

```
$ docker rm 4da79fcb7b39  # 删除编号为 4da79fcb7b39 的容器
```

```
xyj@xyj-home:~$ docker ps                       首先需要确认被删除的容器是否在运行
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
b9c54934f02e        busybox             "sh"                About an hour ago   Up 42 minutes                           musing_murdock
xyj@xyj-home:~$ docker stop b9c54934f02e 停止容器
b9c54934f02e
xyj@xyj-home:~$ docker ps -a                    查看全部容器状况
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
b9c54934f02e        busybox             "sh"                About an hour ago   Exited (137) 26 seconds ago                     musing_murdock   待删除容器
57e348494857        busybox             "sh"                About an hour ago   Exited (0) About an hour ago                    clever_almeida
xyj@xyj-home:~$ docker rm b9c54934f02e   使用容器ID删除，也可以使用别名删除
b9c54934f02e
xyj@xyj-home:~$ docker ps -a                    查看全部容器状况，musing_murdock容器已被删除
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
57e348494857        busybox             "sh"                About an hour ago   Exited (0) About an hour ago                    clever_almeida
```

图 5-32

（11）查看本地所有镜像。

```
$ docker images  # 查看本地存在的镜像，如图 5-33 所示
```

```
xyj@xyj-home:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine              latest              a41a7446062d        2 weeks ago         3.97 MB
busybox             latest              c75bebcdd211        3 weeks ago         1.11 MB
hello-world         latest              48b5124b2768        4 months ago        1.84 kB
roheike/flask-gft   latest              e42a4bc6fd87        6 months ago        275 MB
xyj@xyj-home:~$
拉取路径    镜像版本标签    镜像ID    镜像创建时间    镜像大小
```

图 5-33

（12）删除本地镜像。

当确定不再使用某个镜像时，就可以删除这个镜像，以减少镜像占用空间。删除镜像时一

定要先删除占用这个镜像的容器，即先执行第 9 步，如图 5-34 所示。

```
$ docker rmi roheike/flask-gft  # 删除 roheike/flask-gft 的镜像
```

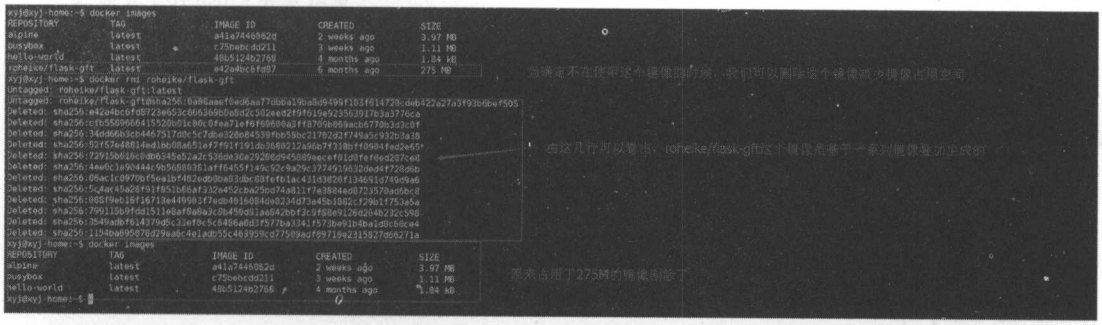


图 5-34

注意：命令 rm 和 rmi 很接近，但是两个命令并不相同，rm 是删除容器，rmi 是删除镜像。

（13）查看本地镜像更改历史。

```
$ docker history hello-world  #查看镜像历史记录，如图 5-35 所示
```

图 5-35

（14）导出镜像

在未搭建 registry 服务器之前，可以将本地镜像导出（tar 包），以方便镜像共享，如图 5-36 所示。

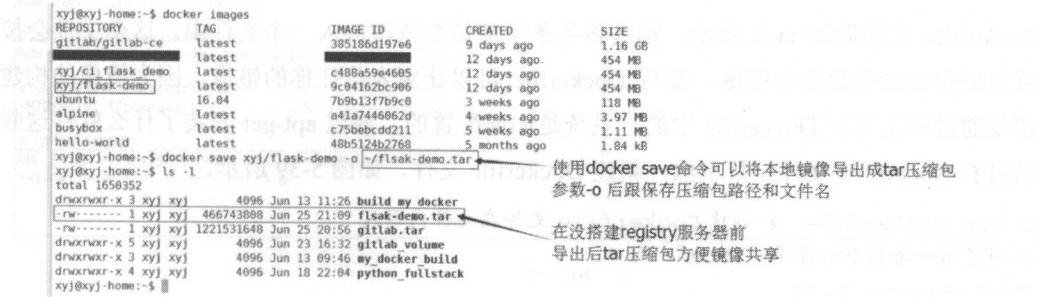


图 5-36

（15）导入镜像。

将导出的 tar 压缩包复制到其他服务器上后，即可使用 Docker load 命令将其导入本地镜像库中，如图 5-37 所示。

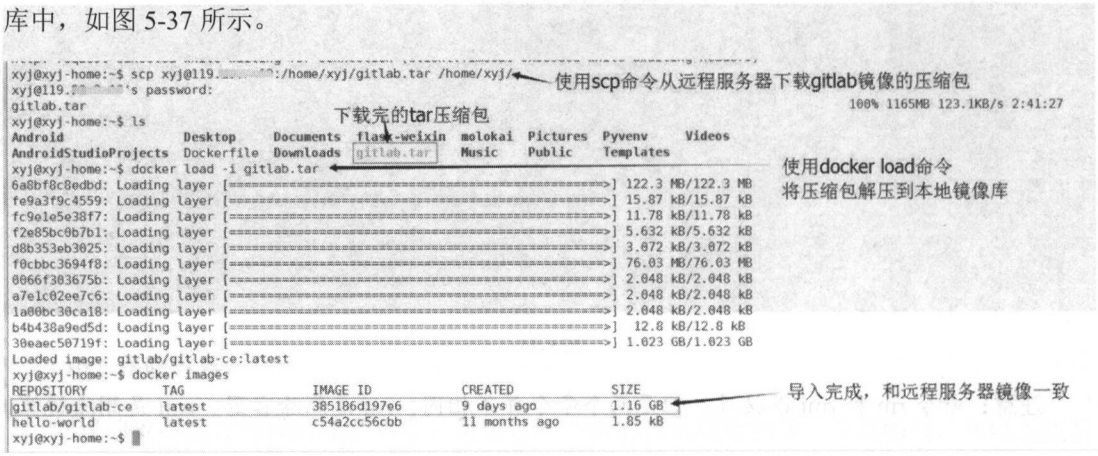


图 5-37

6. 构建私有镜像

（1）新建目录。

```
$ mkdir my_docker_build && cd my_docker_build  # 新建 my_docker_build 目录并
# 进入这个目录，如图 5-38 所示
```

图 5-38

（2）创建 Dockerfile 文件。

Dockerfile 文件类似 shell 脚本，都是将各条所需指令依次放入一个文件中。这个文件会按命令执行顺序自动构建设定镜像。编写 Dockerfile 可以让别人信任你的镜像，因为镜像的构建过程都是通过明文写在 Dockerfile 中的。系统是如何配置的、通过 apt-get 安装了什么软件这些都能通过看 Dockerfile 文件一目了然。新建 Dockerfile 文件，如图 5-39 所示。

```
$ vim Dockerfile  # 创建 Dockerfile 文件在文件内写入
# 规定第一条指令必须是“FROM”
FROM ubuntu:16.04
# 作者，原来“MAINTAINER”已被废弃，所以使用了 LABEL 代替
```

```
LABEL maintainer "xieyingjun@vip.qq.com"
# 运行 apt 更新软件包列表
RUN apt-get update -y
# 安装 Python3 开发环境
RUN apt-get install -y python3-pip python3-dev
# 复制当前目录下的 app 文件夹到容器的/app 目录
COPY ./app /app
# 容器暴露一个 8080 端口，最好不要在 Dockerfile 中配置，以免集群管理时端口冲突，最好在
# 镜像运行时通过命令去绑定端口
# EXPOSE 8080
# 设置容器 shell 运行的工作目录
WORKDIR /app
# 运行升级 pip 版本
RUN pip3 install --upgrade pip
# 运行 pip3 安装 requirements 中依赖包
RUN pip3 install -r requirements.txt
# 设置容器内默认启动的应用
ENTRYPOINT ["python3"]
# 运行 flask 脚本
CMD ["app.py"]
```

```
xyj@xyj-home:~/my_docker_build$ cat Dockerfile
# 规定第一条指令必须是"FROM"
FROM ubuntu:16.04

# 作者,原来"MAINTAINER"已被废弃,所以使用了LABEL代替
LABEL maintainer "xieyingjun@vip.qq.com"

# 运行apt更新软件包列表
RUN apt-get update -y

# 安装python3开发环境
RUN apt-get install -y python3-pip python3-dev

# 复制当前目录下的app文件夹到容器的/app目录
COPY ./app /app

# 设置容器shell运行的工作目录
WORKDIR /app

# 运行升级pip版本
RUN pip3 install --upgrade pip

# 运行pip3安装requirements中依赖包
RUN pip3 install -r requirements.txt

# 设置容器内默认启动的应用
ENTRYPOINT ["python3"]

# 运行flask脚本
CMD ["app.py"]xyj@xyj-home:~/my_docker_build$
xyj@xyj-home:~/my_docker_build$
```

新建Dockerfile文件，内写入内容，创建一个flask示例

图 5-39

（3）创建 flask 示例。

这里我们用 flask 最基本的示例来演示容器运行后提供的网站服务，新建 app 文件夹，如图 5-40 所示。

```
$ mkdir app && cd app # 创建 app 目录并进入该目录
```

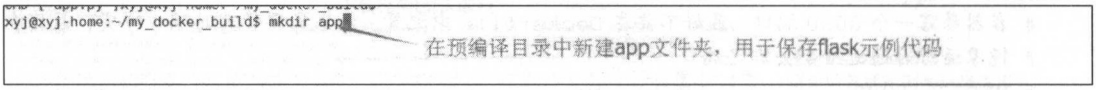


图 5-40

```
$ vim app.py  # 新建 app.py 脚本，写入下面代码
#!/usr/bin/env python3
# encoding: utf-8
"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/4/24
flask 简单示例
"""
from flask import Flask
app = Flask(__name__)
@app.route('/')
def index():
    return "Hello World!"
if __name__=='__main__':
    app.run(debug=True, host="0.0.0.0", port=8080)
$ pip3 freeze >> requirements.txt  # 将示例依赖包导出到 requirements.txt 文件
```

用 Dockerfile 文件构建 flask 示例，如图 5-41 所示。

（4）构建镜像

用命令 Docker build 去自动构建镜像，默认情况下会找当前目录下的 Dockerfile 文件，可以通过-f <path/file>参数控制 Dockerfile 文件的名字和路径，但最好是在一个空白的目录下编译镜像，把需要放入镜像的文件夹集中存放在该目录下。参数-t <images_tags>是给镜像打上标签，通过 Docker images 可以看到。

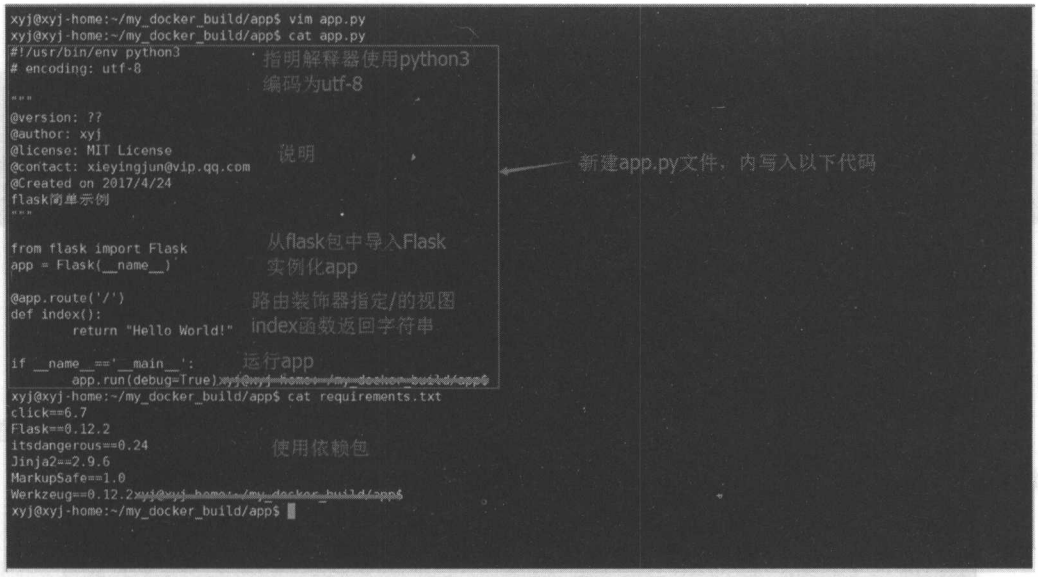


图 5-41

```
$ docker build -t xyj/flask-demo .  # 开始自动构建镜像，注意最后的“.”
```

自动构建镜像如图 5-42 所示。

图 5-42

查看正在构建镜像的本地镜像库如图 5-43 所示。

图 5-43

构建镜像过程如图 5-44 所示，每一步对应 Dockerfile 中的每一行。

```
xyj@xyj-home:~/my_docker_build$ docker build -t xyj/flask-demo .
Sending build context to Docker daemon  5.12 kB
Step 1/11 : FROM ubuntu:16.04
16.04: Pulling from library/ubuntu
bd97b43c27e3: Pull complete
6960dc1aba18: Pull complete
2b61829b0db5: Pull complete
1f88dc826b14: Pull complete
73b3859b1e43: Pull complete
Digest: sha256:ea1d854d38be82f54d39efe2c67000bed1b03348bcc2f3dc094f260855dff368
Status: Downloaded newer image for ubuntu:16.04
 ---> 7b9b13f7b9c0
Step 2/11 : LABEL maintainer "xieyingjun@vip.qq.com"
 ---> Running in 9014567bfb14
 ---> 94d2989f52ac
Removing intermediate container 9014567bfb14
Step 3/11 : RUN apt-get update -y
 ---> Running in b0cf9998edba
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
...(省略连接网络列表)
Fetched 19.3 MB in 1h 1min 55s (5184 B/s)
Reading package lists...
 ---> efca46d78acb
Removing intermediate container b0cf9998edba
Step 4/11 : RUN apt-get install -y python3-pip python3-dev
 ---> Running in d12a6d19a50a
Reading package lists...
Building dependency tree...
Reading state information...
...(省略安装过程)
173 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...
done.
 ---> 269d14307aac
Removing intermediate container d12a6d19a50a
Step 5/11 : COPY ./app /app
 ---> 04377e6d5a00
Removing intermediate container 18d20cefbbf8
Step 6/11 : EXPOSE 8080
 ---> Running in e74b4ed16ef9
 ---> 0a43ce6fdb90
Removing intermediate container e74b4ed16ef9
Step 7/11 : WORKDIR /app
 ---> 20ebda35a921
Removing intermediate container 4f9d38c7420c
Step 8/11 : RUN pip3 install --upgrade pip
 ---> Running in 5e89232a860f
...(省略pip升级过程)
 ---> 06c000bf78e7
Removing intermediate container 5e89232a860f
Step 9/11 : RUN pip3 install -r requirements.txt
 ---> Running in 531b6fabc873
...(省略依赖安装过程)
 ---> d8afc5a2b045
Removing intermediate container 531b6fabc873
Step 10/11 : ENTRYPOINT python3
 ---> Running in 53f4d2fea0dd
 ---> 4e931a6ac349
Removing intermediate container 53f4d2fea0dd
Step 11/11 : CMD app.py
 ---> Running in d5152362d5b4
 ---> 30981311135f
Removing intermediate container d5152362d5b4
Successfully built 30981311135f
```

图 5-44

镜像构建完成后，查看本地镜像，如图 5-45 所示。

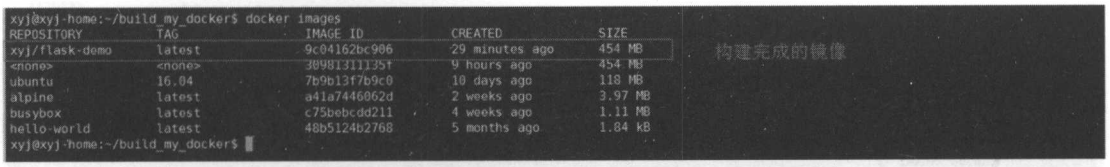


图 5-45

（5）运行镜像。

前面我们构建完成的镜像在本地镜像库中可以找到，但是这个镜像没有被容器运行，所以现在我们需要用这个镜像生成容器。运行镜像如图 5-46 所示。

```
$ docker run -d -p 8080:8080 xyj/flask-demo  # 运行 xyj/flask-demo 这个
#镜像，参数-p 是指宿主机上开启 8080 端口与容器 8080 相互绑定
```

图 5-46

外网访问宿主机外网端口如图 5-47 所示。

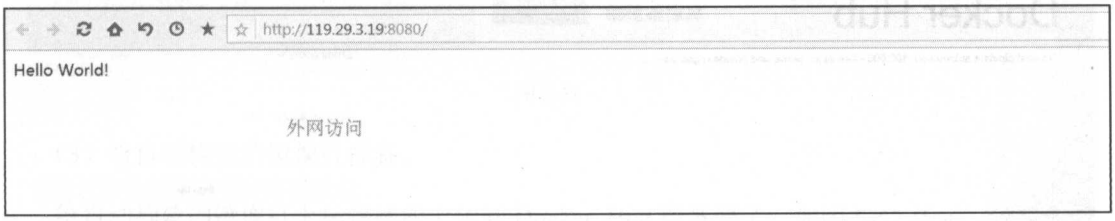


图 5-47

7. 将生成镜像提交到公共库

（1）注册 Docker 账号。

假如已注册了 Docker 账号则可以跳过这一步。打开 https://hub.Docker.com/网址，按照图 5-48 所示输入资料信息，完成注册。

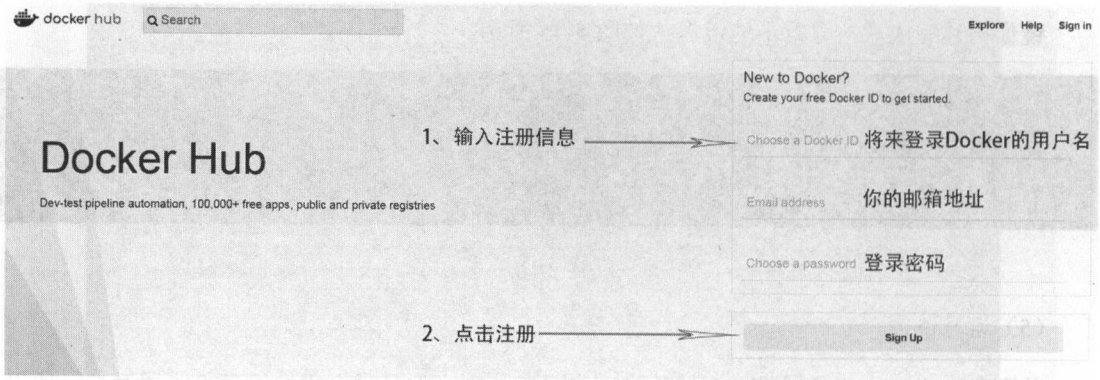


图 5-48

（2）登录 Docker 账号并新建镜像库。

打开 https://hub.Docker.com/网址，单击右上角的“Sign in”跳转到登录页面，如图 5-49 所示。

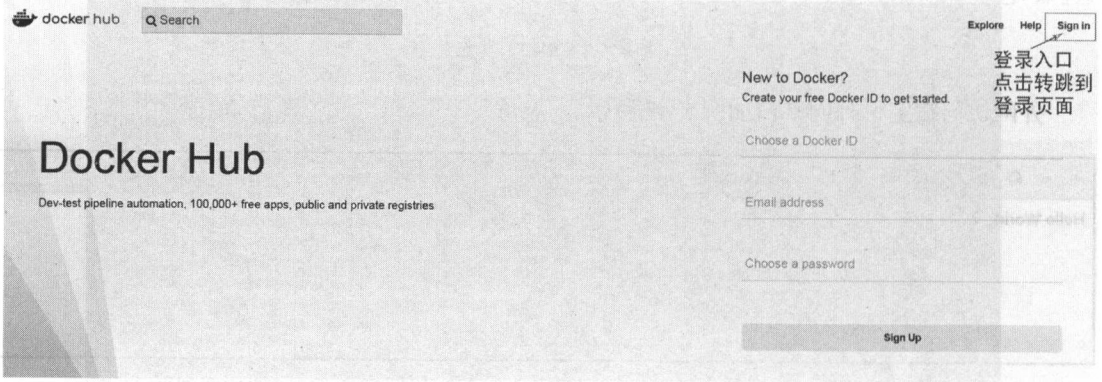


图 5-49

登录后，就可以创建镜像库了，免费用户可以创建一个私有镜像库和一个镜像库，如图 5-50 和图 5-51 所示，共享是免费的，与 GitHub 类似，私有付费。

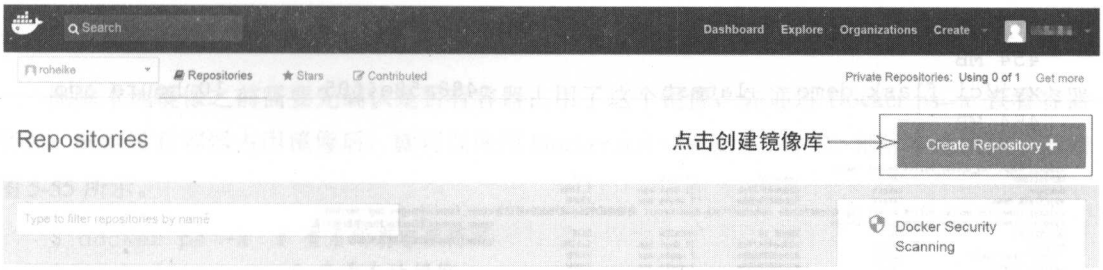


图 5-50

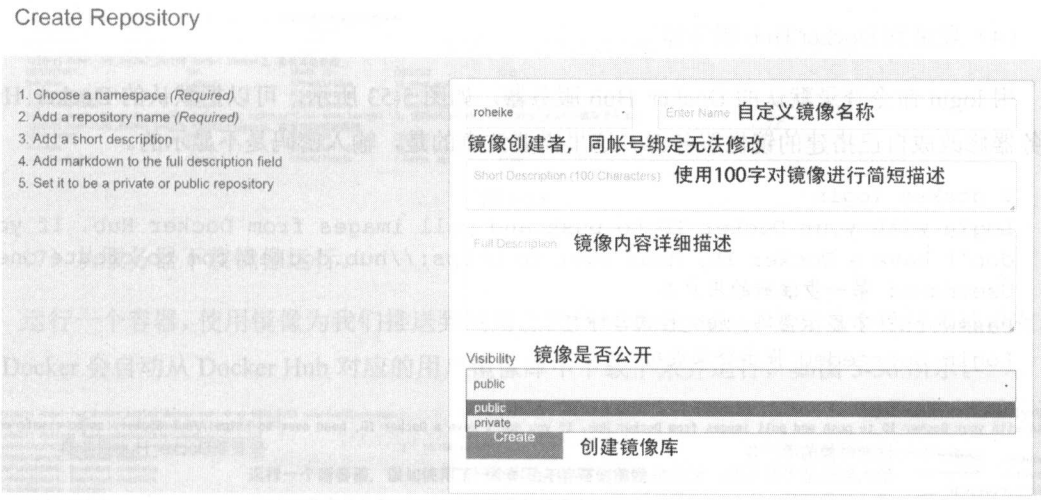


图 5-51

（3）给自动构建的镜像打标签。

给自动构建的镜像打上标签就能上传到 Docker Hub 服务器了，如图 5-52 所示。这里标签名字需要符合：注册账号名字/镜像库名称：镜像版本。

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
xyj/ci_flask_demo   latest              c488a59e4605        10 hours ago
454 MB
$ docker tag c488a59e4605 ****/flask-demo:latest  # 将镜像 IDc488a59e4605
# 标记为<user_name>/images_tag:<version>
$ docker images  # 查看本地镜像
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
```

```
****/flask-demo   latest              c488a59e4605        10 hours ago
454 MB
xyj/ci_flask_demo    latest              c488a59e4605        10 hours ago
454 MB
```

```
xyj@xyj-home:~/my_docker_build$ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
busybox                     latest              d20ae45477cb        12 days ago         1.13MB
xyj/flask-demo              latest              926d4878d69a        7 weeks ago         733MB
xyj@xyj-home:~/my_docker_build$ docker tag 926d4878d69a ****/flask-demo:latest 将镜像ID926d4878d69a标记为<user_name>/images_tag:<version>
xyj@xyj-home:~/my_docker_build$ docker images                                  <user_name>为Docker Hub中的用户名
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE        images_tag为自定义镜像库名称
busybox                     latest              d20ae45477cb        12 days ago         1.13MB
****/flask-demo             latest              926d4878d69a        7 weeks ago         733MB
xyj/flask-demo              latest              926d4878d69a        7 weeks ago         733MB
```

图 5-52

（4）登录到 Docker Hub 服务器。

用 login 命令登录默认的 Docker Hub 服务器，如图 5-53 所示，可以把默认的 Docker Hub 服务器修改成自己搭建的镜像服务器。这里需要注意的是，输入密码是不显示的。

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you
don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: 第一步注册的用户名
Password: 不显示密码，输完按回车即可
Login Succeeded 提示登录成功
```

```
xyj@xyj-home:~$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one
.                                                                                                    登录到Docker Hub服务器
Username:  注册时候的用户名
Password:  输入密码是不显示的，所以按你的密码输完回车就行了
Login Succeeded 提示登录成功
xyj@xyj-home:~$
```

图 5-53

（5）提交镜像。

使用 push 命令将通过 Dockerfile 提交镜像到 Docker Hub 服务器，如图 5-54 所示。

```
$ docker push ****/flask-demo  # 前面一步给镜像打的标签名
```

```
xyj@xyj-home:~$ docker push      /flask-demo                                    使用push命令提交镜像到docker hub服务器
The push refers to a repository [docker.io/      /flask-demo]
e6fbd5c21a30: Pushed
5345af272527: Pushed
77a65c5943b6: Pushed
955eef52a58e: Pushed
625966c1408f: Pushing [=>                                                  ] 10.91 MB/284.2 MB
38016430a2f2: Pushing [==========>                                         ] 11.83 MB/38.44 MB
d8b353eb3025: Pushed
f2e85bc0b7b1: Mounted from library/ubuntu
fc9e1e5e38f7: Mounted from library/ubuntu
fe9a3f9c4559: Mounted from library/ubuntu
6a8bf8c8edbd: Mounted from library/ubuntu
```

这里是上一步定义的镜像标签

图 5-54

（6）删除本地镜像。

删除本地镜像之前需要先确认是否有容器占用了这个镜像，先通过 Docker ps -a 查看容器情况。确定没有容器占用镜像后，就可以使用 Docker rmi <镜像所在库名>删除本地镜像了，如图 5-55 所示。

```
$ docker ps -a  # 查看所有容器情况
$ docker images  # 查看本地镜像
$ docker rmi ****/flask-demo  # 删除本地镜像
```

图 5-55

（7）从服务器下载镜像运行。

运行一个容器，使用镜像为我们推送到网络上的****/flask-demo。如果这个镜像本地没有，则 Docker 会自动从 Docker Hub 对应的用户镜像库中下载下来并运行，如图 5-56 所示。

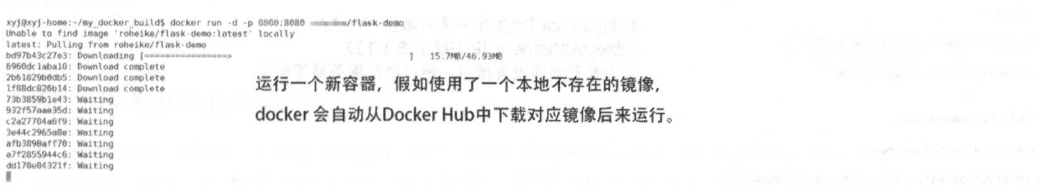


图 5-56

8. 搭建私有 Docker Hub 服务器

（1）拉取 registry 镜像。

```
$ docker pull registry  # 拉取官方 registry 镜像，如图 5-57 所示
```

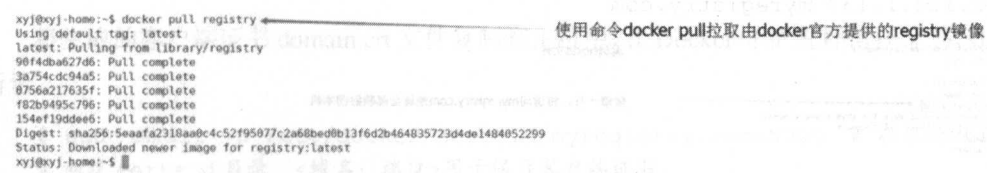


图 5-57

（2）创建用自签名证书。

Docker 出于安全考虑，强制生产环境下的 registry 必须受到 TLS 保护，默认启动的 registry 只能使用 localhost 方式访问。而 TLS 证书分为两种：一种是向专业的认证公司申请，需要付费，申请到的证书会自动添加到浏览器的信任区中；另外一种是我们自行生成的证书，只是用于加密通信，但是浏览器会警告不安全。所以这里我们使用 openssl 制作自己的自签名证书（Self Signed）。

① 修改 ssl 配置文件。

由于是使用自签名证书，所以需要修改 ssl 配置，如图 5-58 所示。

```
$ sudo vim /etc/ssl/openssl.cnf  # 修改 ssl 配置
subjectAltName = IP:192.168.1.137  # 新增本机 IP
```

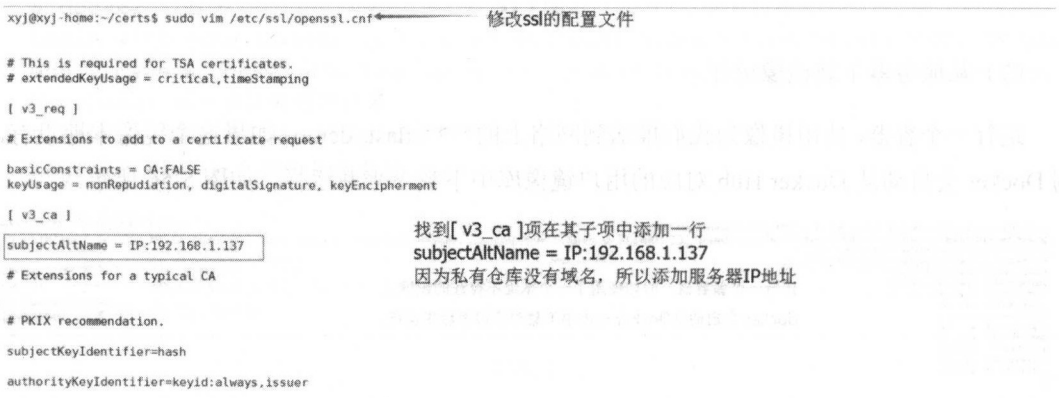


图 5-58

② 修改 hosts 文件。

修改 hosts 文件，使访问 myregistry.com 域名的人可直接访问本机，如图 5-59 所示。

```
$ sudo vim /etc/hosts
192.168.1.137 myregistry.com
```

图 5-59

③ 创建私钥。

```
$ openssl genrsa -out domain.key 4096  # 创建一个使用 4096 位的私钥 domain.key
#文件，如图 5-60 所示
```

```
xyj@xyj-home:~/certs$ openssl genrsa -out domain.key 4096 ←——— 使用openssl创建一个4096位私钥
Generating RSA private key, 4096 bit long modulus
.....++
.....++
e is 65537 (0x10001)
xyj@xyj-home:~/certs$
```

图 5-60

④ 生成证书请求文件 CSR。

```
$ openssl req -new -key domain.key -out domain.csr  # 使用 openssl req 创建
#证书请求，使用 domain.key 私钥生成 domain.csr 文件，如图 5-61 所示
```

```
xyj@xyj-home:~/certs$ openssl req -new -key domain.key -out domain.csr ←——— 使用openssl req创建证书请求csr文件
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CN ←——— 所属国家
State or Province Name (full name) [Some-State]:GD ←——— 所属省份
Locality Name (eg, city) []:FS ←——— 所属地市
Organization Name (eg, company) [Internet Widgits Pty Ltd]:GFT ←——— 所属公司
Organizational Unit Name (eg, section) []:JSB ←——— 所属部门
Common Name (e.g. server FQDN or YOUR name) []:myregistry.com ←——— 重点：需要指定一个域名或主机名
Email Address []:XIEYINGJUN@VIP.QQ.COM ←——— 邮箱

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:LANMAO ←——— 加密证书请求的密码
An optional company name []:
```

图 5-61

⑤ 生成 Self Signed 证书。

```
$ openssl x509 -req -days 365 -in domain.csr -signkey domain.key -out
domain.crt  # 创建 Self Signed 证书，使用 domain.key 私钥生成 CA 证书，有效期为 365
#天，如图 5-62 所示
```

```
xyj@xyj-home:~/certs$ openssl x509 -req -days 365 -in domain.csr -signkey domain.key -out domain.crt ←——— 生成Self Signed证书，有效期365天
Signature ok
subject=/C=CN/ST=GD/L=FS/O=GFT/OU=JSB/CN=myregistry.com/emailAddress=XIEYINGJUN@VIP.QQ.COM
Getting Private key
```

图 5-62

⑥ 将客户端认证证书加入主机。

将生成的客户端证书 domain.crt 文件复制到主机中，让 Docker 守护进程信任证书，如图 5-63 所示。

```
$ sudo mkdir -p /etc/docker/certs.d/myregistry.com:5000  # 在 docker 配置中
# 新建 certs.d 目录，<域名：端口>用于保存客户端证书
```

```
$ sudo cp domain.crt /etc/docker/certs.d/myregistry.com:5000/ca.crt  # 将
# 客户端证书复制到 Docker 配置中
$ sudo cp domain.crt /usr/local/share/ca-certificates/myregistry.com.crt
# 将客户端证书添加到系统根证书目录下
$ sudo update-ca-certificates  # 更新系统证书
```

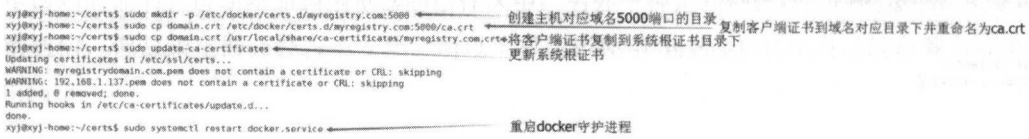


图 5-63

（3）创建数据目录。

```
$ mkdir -p docker-registry_volums  # 创建用于保存镜像数据的目录，与容器绑定
$ mkdir -p docker-registry-auth  # 创建用于保存基础身份验证的密码文件
```

如图 5-64 所示。

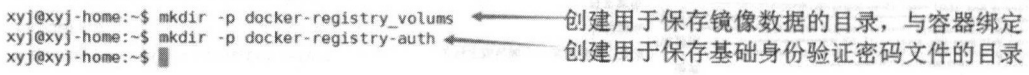


图 5-64

（4）启动 registry 容器。

```
docker run -d \  # 后台启动服务
  --name c-registry \  # 容器别名
  -v /home/xyj/certs:/certs \  # 证书映射
  -e REGISTRY_HTTP_ADDR=0.0.0.0:8080 \  # 更改 registry 服务端口
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \  # 证书请求文件在容器
# 内路径
  -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \  # 私钥在容器内路径
  -p 8080:8080 \  # 与宿主机端口绑定
  -v /home/xyj/docker-registry_volums:/var/lib/registry \  # 将本地磁盘映射
# 到容器内，用于持久保存镜像数据
  registry  # 启动镜像
```

如图 5-65 所示。

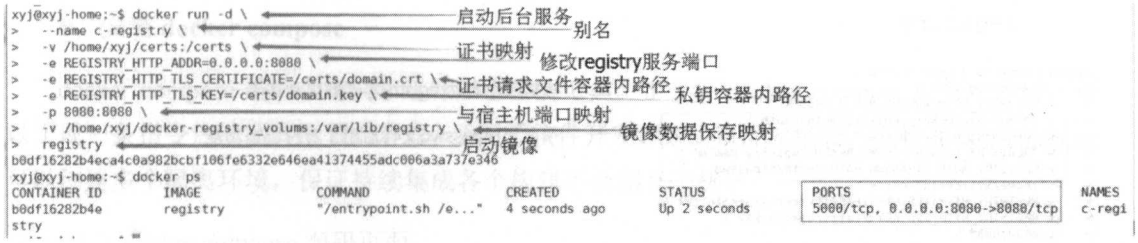


图 5-65

（5）创建基础身份认证密码文件。

```
 $ docker run \  # 运行一个容器
--entrypoint htpasswd \  # 容器内运行 htpasswd 命令
registry -Bbn xyj docker-registry > docker-registry-auth/htpasswd  # 生成
# 用户名 xyj 密码 docker-registry 的信息，保存到宿主机的 docker-registry-auth 目录
# 下的 htpasswd 文件中
```

如图 5-66 所示。

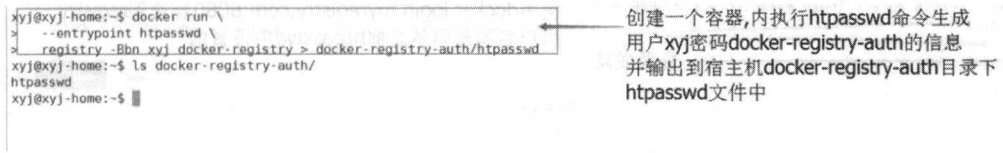


图 5-66

（6）启用身份验证。

这一步与第四步启动 registry 容器的命令的区别在于引入了 htpasswd 文件，启用了身份验证功能，如图 5-67 所示。

```
$ docker run -d \
  --name c-registry \
  -e REGISTRY_HTTP_ADDR=0.0.0.0:8080 \
  -v /home/xyj/docker-registry-auth:/auth \  # 将宿主机的#
docker-registry-auth 目录映射到容器内/auth
  -e "REGISTRY_AUTH=htpasswd" \  # 身份验证 htpasswd 文件
  -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \  #
  -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \  # htpasswd 在容器内路径
  -v /home/xyj/certs:/certs \
  -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
  -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
  -p 8080:8080 \
```

```
    registry
```

```
xyj@xyj-home:~$ docker run -d \
>   --name c-registry \
>   -e REGISTRY_HTTP_ADDR=0.0.0.0:8080 \
>   -v /home/xyj/docker-registry-auth:/auth \
>   -e "REGISTRY_AUTH=htpasswd" \
>   -e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
>   -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
>   -v /home/xyj/certs:/certs \
>   -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
>   -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
>   -p 8080:8080 \
>   registry
5d2e10a0fc39e14196c402bc3277767ffcd425e231e9f750c9848d8da7a52362
xyj@xyj-home:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                              NAMES
5d2e10a0fc39        registry            "/entrypoint.sh /e..."   4 seconds ago       Up 3 seconds        5000/tcp, 0.0.0.0:8080->8080/tcp   c-regi
stry
xyj@xyj-home:~$
```

与第四步启动registry容器的区别在于

引入了htpasswd文件启用了身份验证功能

图 5-67

（7）登录到 registry。

使用 docker login <server:port>登录到 registry 如图 5-68 所示。

```
$ docker login myregistry.com:8080  # 登录到本机的 registry 中
```

```
xyj@xyj-home:~/certs$ docker login myregistry.com:8080
Username: xyj
Password:
Login Succeeded
```

使用docker login myregistry.com:8080登录到registry

用户名和密码是之前htpasswd中设置的

登录成功

图 5-68

（8）提交镜像。

前面是提交镜像到 hub.docker.com，现在我们提交镜像到本地仓库，如图 5-69 所示。

```
$ docker tag hello-world myregistry.com:8080/my-hello  # 将 hello-world 镜
#像打标签为本地仓库镜像
$ docker push myregistry.com:8080/my-hello  # 提交镜像
```

```
xyj@xyj-home:~/certs$ docker images
REPOSITORY                      TAG                 IMAGE ID            CREATED             SIZE
c-gitbal                        latest              87a060834064        25 hours ago        1.24 GB
xyj/gitlab                      latest              87a060834064        25 hours ago        1.24 GB
gitlab/gitlab-runner-helper     x86_64-3df822b      fc6b1b9db456        2 days ago          50.5 MB
intellij/docker/compose         1.11.2              678d53b4c9a5        3 days ago          59.4 MB
pycharm_helpers                 PY-171.4694.38      ec5a5818b288        3 days ago          25.8 MB
flaskdemo_web                   latest              3c8a3e13a0dd        4 days ago          694 MB
flaskcompose_web                latest              46ebc876c613        4 days ago          695 MB
docker                          dind                5096e5a0cba0        9 days ago          103 MB
docker                          latest              192e3edb771f        9 days ago          97 MB
registry                        2                   c2a449c9f834        10 days ago         33.2 MB
registry                        latest              c2a449c9f834        10 days ago         33.2 MB
postgres                        latest              5fa273ee7568        2 weeks ago         269 MB
mysql                           latest              44a8e1a5c8b2        2 weeks ago         407 MB
mongo                           latest              71c101e16e61        2 weeks ago         358 MB
gitlab/gitlab-runner            latest              efff6941f780        2 weeks ago         399 MB
python                          3                   f9a9a73a3680        2 weeks ago         684 MB
python                          3.6                 f9a9a73a3680        2 weeks ago         684 MB
gitlab/gitlab-ce                latest              385186d197e6        3 weeks ago         1.16 GB
docker/compose                  1.11.2              61252c282969        4 months ago        59.4 MB
hello-world                     latest              c54a2cc56cbb        12 months ago       1.85 kB
xyj@xyj-home:~/certs$ docker tag hello-world myregistry.com:8080/my-hello
xyj@xyj-home:~/certs$ docker push myregistry.com:8080/my-hello
The push refers to a repository [myregistry.com:8080/my-hello]
a02596fdd012: Pushed
latest: digest: sha256:4c8e63ca4cb663ce6c688cb06f1c3672a172b088dac5b6d7ad7d49cd620d85cf size: 524
xyj@xyj-home:~/certs$
```

将hello-world镜像打标签为myregistry.com:8080/my-hello镜像

使用docker push命令提交镜像

提交完成

图 5-69

9. 安装 docker compose

docker compose 是一个定义和运行多容器的工具。定义好 docker compose 配置文件后，可以实现一条指令启动所有配置的容器，适用于软件开发、测试和持续集成。使用 docker compose 可以创建多个隔离环境，保证持续集成各个构建不会相互干扰。

（1）docker compose 源码页面。

在 Mac 和 Windows 系统下安装 Docker 引擎会自动安装 docker compose，但 Ubuntu 下无集成，所以我们需要手动安装一下。

docker compose 的源码在 GitHub 上面有共享，用浏览器打开 https://github.com/docker/compose/releases，找到最新版本的安装链接，如图 5-70 所示。

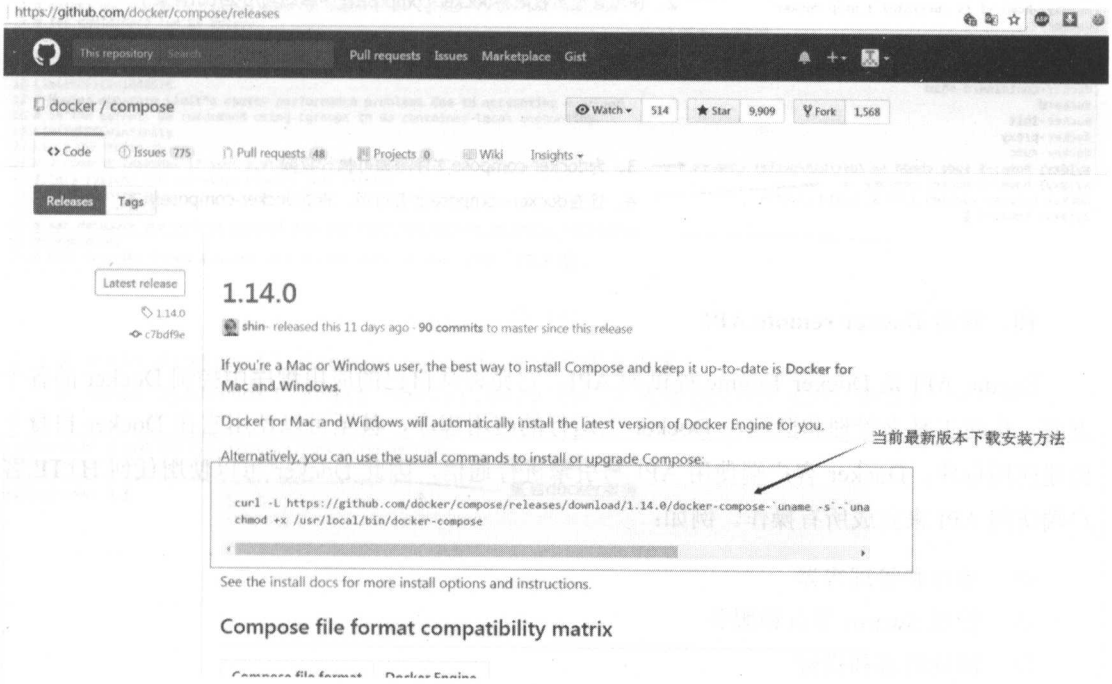


图 5-70

（2）下载 docker compose 并添加到环境变量中。

按照安装提示下载 docker compose 后，在终端中使用管理员权限运行，如图 5-71 所示。

```
$ curl -L
https://github.com/docker/compose/releases/download/1.14.0/docker-compo
se-`uname -s`-`uname -m` > ~/docker-compose  # 下载 docker-compose 程序
$ sudo mv docker-compose /usr/bin/  # 使用管理员权限将 docker-compose 移动到
# /use/bin 目录下
$ sudo chmod +x /usr/bin/docker-compose  # 为 docker-compose 文件添加可执行权
# 限
$ docker-compose -v  # 检查 docker-compose 是否可用，通过命令 docker-compose 查
# 看程序版本
```

图 5-71

10. 开启 Docker remote API

Engine API 是 Docker Engine 提供的 API。它允许从自己的应用程序中控制 Docker 的各个方面，构建工具来管理和监视在 Docker 上运行的应用程序，甚至可以使用它在 Docker 自身上构建应用程序。Docker 客户端使用 API 与引擎进行通信，因此 Docker 可以使用任何 HTTP 客户端访问 API 来完成所有操作。例如：

- ◎ 运行和管理容器
- ◎ 管理 Swarm 节点和服务
- ◎ 阅读日志和指标
- ◎ 创建和管理群集
- ◎ 拉动和管理图像
- ◎ 管理网络和卷

（1）在 Docker Server 文件新增监听端口。

使用 Vim 编辑/lib/systemd/system/Docker.service 文件，增加服务器启动时监听端口，这里设置监听 2376 端口，如图 5-72 所示。

```
$ sudo systemctl stop docker.service  # 停止 Docker 服务
$ sudo vim /lib/systemd/system/docker.service  # 编辑 Docker 服务启动文件
#ExecStart=/usr/bin/dockerd -H fd://  # 在前面添加#号注释原有配置
ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2376 -H
unix://var/run/docker.sock  # 新增监听 2376 端口
```

```
xyj@xyj-OptiPlex-9020:~/Downloads$ sudo vim /lib/systemd/system/docker.service ← 编辑docker服务文件
 1 [Unit]
 2 Description=Docker Application Container Engine
 3 Documentation=https://docs.docker.com
 4 After=network-online.target docker.socket firewalld.service
 5 Wants=network-online.target
 6 Requires=docker.socket
 7
 8 [Service]
 9 Type=notify
10 # the default is not to use systemd for cgroups because the delegate issues still
11 # exists and systemd currently does not support the cgroup feature set required
12 # for containers run by docker
13 #ExecStart=/usr/bin/dockerd -H fd:// ← 注释原有的配置
14 ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2376 -H unix://var/run/docker.sock ← 新输入启动监听2376端口
15 ExecReload=/bin/kill -s HUP $MAINPID
16 LimitNOFILE=1048576
17 # Having non-zero Limit*s causes performance problems due to accounting overhead
18 # in the kernel. We recommend using cgroups to do container-local accounting.
19 LimitNPROC=infinity
20 LimitCORE=infinity
21 # Uncomment TasksMax if your systemd version supports it.
22 # Only systemd 226 and above support this version.
23 TasksMax=infinity
24 TimeoutStartSec=0
25 # set delegate yes so that systemd does not reset the cgroups of docker containers
26 Delegate=yes
27 # kill only the docker process, not all processes in the cgroup
```

图 5-72

```
$ sudo systemctl daemon-reload  # 重新载入配置
$ sudo systemctl start docker.service  # 重启 Docker 服务，如图 5-73 所示
```

```
xyj@xyj-home:~$ sudo vim /lib/systemd/system/docker.service
xyj@xyj-home:~$ sudo systemctl daemon-reload ← 重新加载配置
xyj@xyj-home:~$ sudo systemctl start docker.service ← 重启docker服务
xyj@xyj-home:~$
```

图 5-73

（2）使用 curl 查看 Docker 版本信息。

使用 curl 测试 Docker 服务是否开始监听 2376 端口，如图 5-74 所示。

```
$ curl http://localhost:2376/version
```

图 5-74

（3）API 示例。

① 远程运行容器，如图 5-75 所示。

```
$ curl -H "Content-Type: application/json" \  # 构建一个 http 头
> -d '{"Image": "hello-world"}' \  # 传输 JSON 数据
> -X POST http://192.168.1.137:2376/containers/create  # 使用 POST 提交到远
# 程服务器的/containers/create 路径
```

图 5-75

② 列出当前运行的容器信息，如图 5-76 所示。

直接访问/containers/json 可以获取当前运行的容器信息，相当于 docker ps 命令。

```
$ curl http://192.168.1.137:2376/containers/json
```

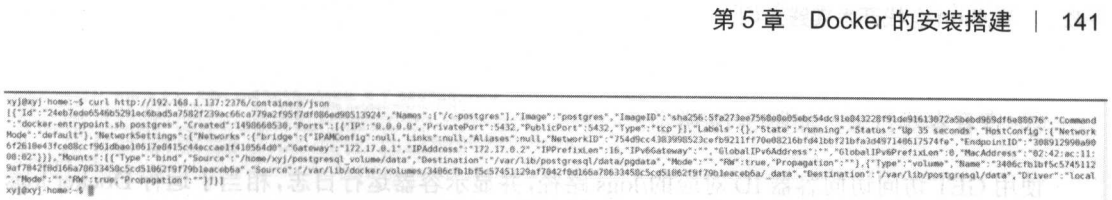


图 5-76

③ 停止容器，如图 5-77 所示。

使用 POST 方式访问容器 ID 对应的/stop 路径，并停止该容器。相当于运行 docker stop <containers_id>命令。

```
$ curl -X POST http://192.168.1.137:2376/containers/24eb7ede6546/stop
```

图 5-77

④ 启动容器，如图 5-78 所示。

使用 POST 访问容器 ID 对应的/start 路径，即可启动容器，相当于运行 docker start <containers_id>命令。

图 5-78

⑤ 查看容器日志，如图 5-79 所示。

使用 GET 访问访问容器 ID 对应的/logs 路径，并显示容器运行日志，相当于运行 Docker logs <containers_id>命令。

图 5-79

更多 HTTP API 信息请参考 https://docs.Docker.com/engine/api/v1.27/。

11. 填坑：服务器断电，启动后无法连接

笔者之前在一台服务器上用 Docker push 命令提交镜像时，不小心服务器断电了，重启服务器后发现 Docker 服务遇到下面提示问题。

```
$ docker ps -a
Cannot connect to the Docker daemon. Is the docker daemon running on this
host?
```

解决过程如下。

（1）检查 Docker 进程情况。

我们可以使用 ps -aux 命令查看系统内当前运行的进程，参数-a 可显示其他用户的进程；参数-u 可显示用户名、CPU 使用率、虚拟内存占用、物理内存占用等状态；参数-x 可显示自己运行的进程。但这样会把所有进程都显示出来，所以我们需要过滤一下信息，可以使用 grep 命令来搜索过滤 ps 中的内容，如图 5-80 所示。

```
$ ps -aux|grep docker  # 用 ps -aux 显示所有用户所有进程，把结果通过管道传递给 grep
#查询过滤
```

图 5-80

注意：上面提的“管道”是 Linux 支持的最初的用 UNIX IPC 形式之一，具有以下特点。

- ◎ 管道是半双工的，数据只能向一个方向流动；
- ◎ 需要双方通信时，需要建立两个管道；
- ◎ 只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；
- ◎ 单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，因为它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在于内存中。
- ◎ 数据的读出和写入：一个进程向管道中写入的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。[1]

可以把管道比喻成自来水管，水从左往右流，而每一个“|”是水阀，水阀可以控制水流大小（过滤数据）或者改变水质（数据转换）。

（2）查看 Docker 服务情况。

用 systemctl status 命令可以查看后台服务详细状态，如图 5-81 所示。

```
$ systemctl status docker.service
```

```
$ systemctl status docker.service
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
   Active: failed (Result: exit-code) since 二 2016-11-22 08:32:28 CST; 36min ago   服务状态为failed（启动失败）
     Docs: https://docs.docker.com
  Process: 7012 ExecStart=/usr/bin/dockerd -H fd:// (code=exited, status=1/FAILURE)
 Main PID: 7012 (code=exited, status=1/FAILURE)
11月 22 08:32:27 xyj-OptiPlex-9020 systemd[1]: Starting Docker Application Container Engine...
11月 22 08:32:27 xyj-OptiPlex-9020 dockerd[7012]: time="2016-11-22T08:32:27.688056800+08:00" level=info msg="libcontainerd: new cont
11月 22 08:32:28 xyj-OptiPlex-9020 dockerd[7012]: time="2016-11-22T08:32:28.748462321+08:00" level=info msg="[graphdriver] using pri
11月 22 08:32:28 xyj-OptiPlex-9020 dockerd[7012]: time="2016-11-22T08:32:28.777019931+08:00" level=fatal msg="Error starting daemon:
11月 22 08:32:28 xyj-OptiPlex-9020 systemd[1]: docker.service: Main process exited, code=exited, status=1/FAILURE
11月 22 08:32:28 xyj-OptiPlex-9020 systemd[1]: Failed to start Docker Application Container Engine.
11月 22 08:32:28 xyj-OptiPlex-9020 systemd[1]: docker.service: Unit entered failed state.
11月 22 08:32:28 xyj-OptiPlex-9020 systemd[1]: docker.service: Failed with result 'exit-code'.
```

图 5-81

1　管道解释来至百度百科
http://baike.baidu.com/link?url=PyBfh_DcfUXE3EIY32QAGBSL6qzzpNBswf1OCr0TmUmaAhzJsD-OBZYs-N1ruq5lrUHvzqAsI-BoJ-uumpdvya#2_2

（3）手动启动 Docker 服务。

上面查询 Docker 服务的状态，发现 Docker 守护进程服务启动失败。下面我们通过手动启动一次，看看 Docker 后台服务能不能正常启动，因为有可能是某个配置导致系统不会自动启动 Docker 后台服务，如图 5-82 所示。

```
$ sudo systemctl start docker.service
Job for docker.service failed because the control process exited with error
code. See "systemctl status docker.service" and "journalctl -xe" for details.
```

```
$ sudo systemctl start docker.service  我们重新手工启动一次docker服务，无法启动报错
Job for docker.service failed because the control process exited with error code.
See "systemctl status docker.service" and "journalctl -xe" for details.
$ systemctl status docker.service
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
   Active: failed (Result: exit-code) since 二 2016-11-22 08:43:17 CST; 36min ago  查看服务状态，还是failed启动失败
     Docs: https://docs.docker.com
  Process: 7012 ExecStart=/usr/bin/dockerd -H fd:// (code=exited, status=1/FAILURE)
 Main PID: 7012 (code=exited, status=1/FAILURE)
11月 22 08:42:48 xyj-OptiPlex-9020 systemd[1]: Starting Docker Application Container Engine...
11月 22 08:42:48 xyj-OptiPlex-9020 dockerd[7012]: time="2016-11-22T08:42:48.238016500+08:00" level=info msg="libcontainerd: new cont
11月 22 08:43:17 xyj-OptiPlex-9020 dockerd[7012]: time="2016-11-22T08:43:17.342362334+08:00" level=info msg="[graphdriver] using pr
11月 22 08:43:17 xyj-OptiPlex-9020 dockerd[7012]: time="2016-11-22T08:43:17.827839021+08:00" level=fatal msg="Error starting daemon
11月 22 08:43:17 xyj-OptiPlex-9020 systemd[1]: docker.service: Main process exited, code=exited, status=1/FAILURE
11月 22 08:43:17 xyj-OptiPlex-9020 systemd[1]: Failed to start Docker Application Container Engine.
11月 22 08:43:17 xyj-OptiPlex-9020 systemd[1]: docker.service: Unit entered failed state.
11月 22 08:43:17 xyj-OptiPlex-9020 systemd[1]: docker.service: Failed with result 'exit-code'.
```

图 5-82

（4）查看 systemd 日志。

前面我们手动启动了 Docker 后台服务，查询服务状态后发现还是启动失败，启动时提示我们可以通过 journalctl -xe 查看，如图 5-83 所示。

```
$ journalctl -xe
...
11 月 22 10:45:17 xyj-home dockerd[8453]:
time="2016-11-22T10:45:17.889095348+08:00" level=fatal msg="Error
starting daemon: layer does not exist"
11 月 22 10:45:17 xyj-home systemd[1]: docker.service: Main process exited,
code=exited, status=1/FAILURE
11 月 22 10:45:17 xyj-home systemd[1]: Failed to start Docker Application
Container Engine.
...
```

```
$ journalctl -xe
...(日志略部分无关内容)
-- docker.socket 单元已开始启动。
11月 22 10:45:16 xyj-OptiPlex-9020 systemd[1]: Listening on Docker Socket for the API.
-- Subject: docker.socket 单元已结束启动
-- Defined-By: systemd
-- Support: http://lists.freedesktop.org/mailman/listinfo/systemd-devel
--
-- docker.socket 单元已结束启动。  1、docker.socket单元一启动就关闭了
--
-- 启动结果为"done"。
11月 22 10:45:16 xyj-OptiPlex-9020 systemd[1]: Starting Docker Application Container Engine...
-- Subject: docker.service 单元已开始启动
-- Defined-By: systemd
-- Support: http://lists.freedesktop.org/mailman/listinfo/systemd-devel
--
-- docker.service 单元已开始启动。
11月 22 10:45:16 xyj-OptiPlex-9020 dockerd[8453]: time="2016-11-22T10:45:16.829663347+08:00" level=info msg="libcontainerd: new containerd process, pid:
11月 22 10:45:17 xyj-OptiPlex-9020 audit[8474]: AVC apparmor="STATUS" operation="profile_replace" profile="unconfined" name="docker-default" pid=8474 co
11月 22 10:45:17 xyj-OptiPlex-9020 kernel: audit: type=1400 audit(1479782717.856:46): apparmor="STATUS" operation="profile_replace" profile="unconfined"
11月 22 10:45:17 xyj-OptiPlex-9020 dockerd[8453]: time="2016-11-22T10:45:17.864534925+08:00" level=info msg="[graphdriver] using prior storage driver \"
11月 22 10:45:17 xyj-OptiPlex-9020 dockerd[8453]: time="2016-11-22T10:45:17.889095348+08:00" level=fatal msg="Error starting daemon: layer does not exis
11月 22 10:45:17 xyj-OptiPlex-9020 systemd[1]: docker.service: Main process exited, code=exited, status=1/FAILURE
11月 22 10:45:17 xyj-OptiPlex-9020 systemd[1]: Failed to start Docker Application Container Engine.
-- Subject: docker.service 单元已失败
-- Defined-By: systemd
-- Support: http://lists.freedesktop.org/mailman/listinfo/systemd-devel
--
-- docker.service 单元已失败。
```

2、报错内容:镜像的层找不到

3、因为镜像层丢失所以导致docker引擎退出

图 5-83

注意：journald 是 systemd 独有的日志系统，替换了 sysVinit 中的 syslog 守护进程。命令 journalctl 可用来读取日志。

（5）解决方法。

从上面日志错误提示中可以看到，因为镜像层丢失了，所以导致 Docker 引擎服务启动失败。知道了问题根源所在，这样我们就能对症下药解决故障了。这里用了最暴力的方法——直接删除所有镜像配置。

```
$ sudo rm -rf /var/lib/docker/  # 删除 Docker 的所有配置
$ sudo systemctl start docker.service  # 重启 Docker 服务
```

第6章
Git使用

6.1 版本控制简介

在开发过程中，经常会遇到一个项目由多人合力完成这种情况，每个人负责其中一个模块。项目开发过程中为了确保代码的可追溯，我们引入了版本控制概念，每个人修改了什么代码或提交了什么代码都能够跟踪记录。

现在流行的版本控制主要有：集中式版本控制（SVN）和分布式版本控制（GIT）。

集中式版本控制，顾名思义，项目代码都集中在服务器上面，所有人都要连接服务器才能提交代码，断网后就不能提交代码。

分布式版本控制，分布式与集中式的最大区别在于开发者可以提交到本地，每个开发者通过克隆（git clone），在本地机器上复制一个完整的 Git 仓库[1]。每个人都有一个完整的代码库，完全按照自己的想法开发分支等，然后推送给别人更新合并代码。

6.2 Git 历史

自 2002 年开始，林纳斯·托瓦兹[2]决定使用 BitKeeper 作为 Linux 内核的主要版本控制系统

---

1 引用百度百科解释
http://baike.baidu.com/link?url=Icp6reKQU0KFxQTuja1jn34R1wCNtPWDjtjA2IMnS6_O-q59O3CGG0q1yoUMmXirvdTnkL5dFrK-KrX1Ix63r_

2 林纳斯·本纳第克特·托瓦兹（Linus Torvalds）是 Linux 系统之父，Git 开发者

用以维护代码。由于 BitKeeper 为专有软件，因此这个决定在社区中长期遭受质疑。在 Linux 社区中，特别是理查德 • 斯托曼[1]与自由软件基金会的成员，主张应该使用开放源代码的软件来作为 Linux 的核心版本控制系统。林纳斯 •托瓦兹曾考虑过采用现成软件作为版本控制系统（例如 Monotone），但这些软件都存在一些问题，即性能不佳。现成的方案，如 CVS 的架构，受到林纳斯 • 托瓦兹的批评。

2005 年，安德鲁 • 垂鸠[2]写了一个简单程序，可以连接 BitKeeper 的存储库。BitKeeper 著作权拥有者拉里 • 麦沃伊认为安德鲁 • 垂鸠对 BitKeeper 内部使用的协议进行了逆向工程，因此决定收回无偿使用 BitKeeper 的授权。Linux 内核开发团队与 BitMover 公司进行蹉商，但无法解决他们之间的歧见。林纳斯 • 托瓦兹决定自行开发版本控制系统来替代 BitKeeper，以十天的时间，编写出第一个 Git 版本。[3]

6.3　安装 Git

1. Windows 下安装 Git

相对 Linux 而言，在 Windows 下使用 Git 略烦琐，需要自己安装 MinGW 来实现后 Windows 下使用 Linux 环境，而最简单的方式是安装 git for windows，它自动集成了 MinGW，从而使得 Git 的安装简单化。

（1）下载安装包。

安装包的下载网址是 https://git-for-windows.github.io/，打开该网址，单击“Download”按钮，下载 Git 安装包，如图 6-1 所示。

1　理查德 • 斯托曼（Richard Matthew Stallman）是自由软件基金会的主席

2　安德鲁 • 垂鸠（Andrew Tridgell）是 Samba 的作者

3　引用至 wikipedia https://zh.wikipedia.org/wiki/Git#.E6.AD.B7.E5.8F.B2

图 6-1

（2）Git 运行安装包。

下载好安装包后，直接双击开始安装，如图 6-2 所示。

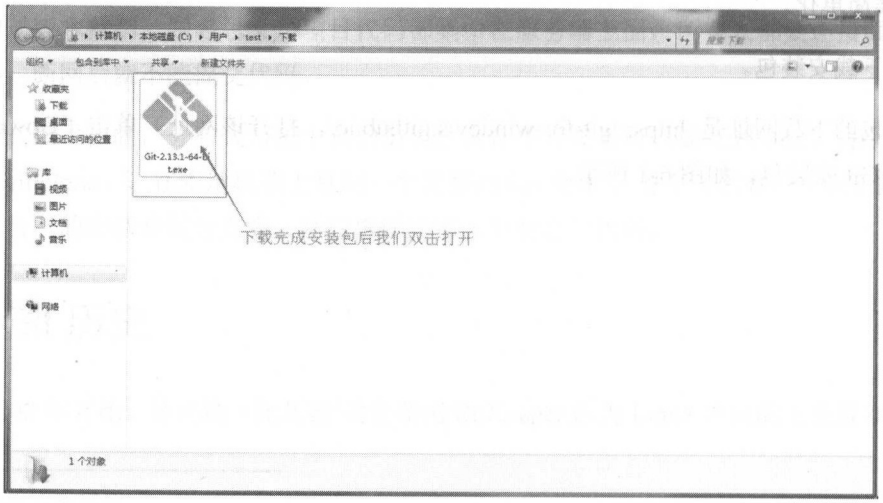


图 6-2

（3）授权协议。

Git 使用 GNU 授权协议，直接单击“NEXT”按钮安装即可，如图 6-3 所示。

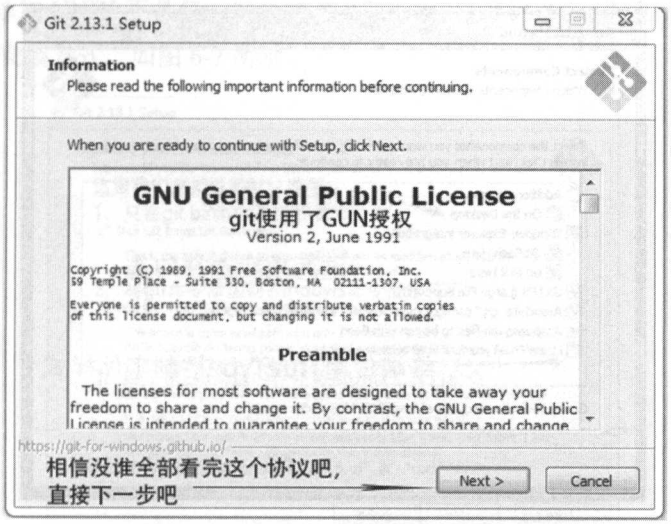


图 6-3

（4）选择安装目录。

这一步是选择安装目录，一般情况下都是默认安装路径，如图 6-4 所示。

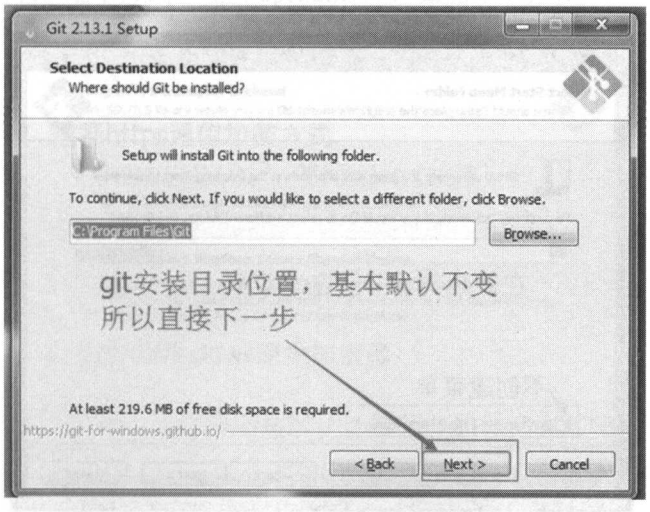


图 6-4

（5）安装选项。

这里我们选择在桌面增加快捷方式和使用 TrueType 控制终端字体样式，如图 6-5 所示。

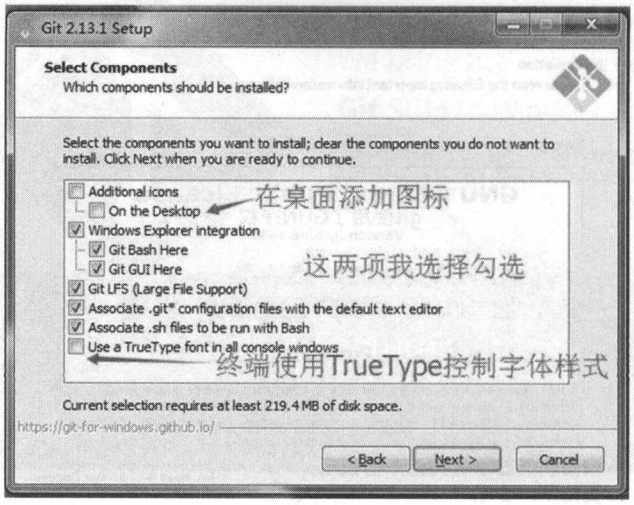


图 6-5

（6）创建菜单。

默认会在开始菜单中创建 Git 菜单，也可以选择不创建，这里选择默认创建，如图 6-6 所示。

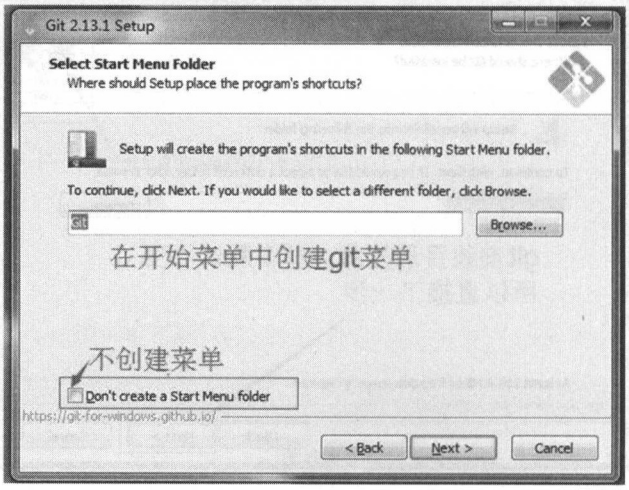


图 6-6

（7）添加环境变量。

选择默认将 Git 命令添加到命令提示符下，第一项只在 Git Bash 中使用 Git 命令；第三项是添加 Git 命令同时添加 UNIX 工具包，但会覆盖 Windows 自带的“find”查找和“sort”排序命令。这里选择默认安装方式，如图 6-7 所示。

图 6-7

（8）选择 https 通信加密方式。

这里选择默认的加密通信方式，使用 openSSL 库，如图 6-8 所示。

图 6-8

（9）换行符配置。

这项配置是为代码库中的换行符进行自动转换，由于 Windows 下的换行符和 Linux 换行符不同，所以需要进行转换。默认推荐检出代码时自动将代码中的换行符转换成符合 Windows 风格的换行符，提交代码时又自动转换成 UNIX 风格的换行符，如图 6-9 所示。

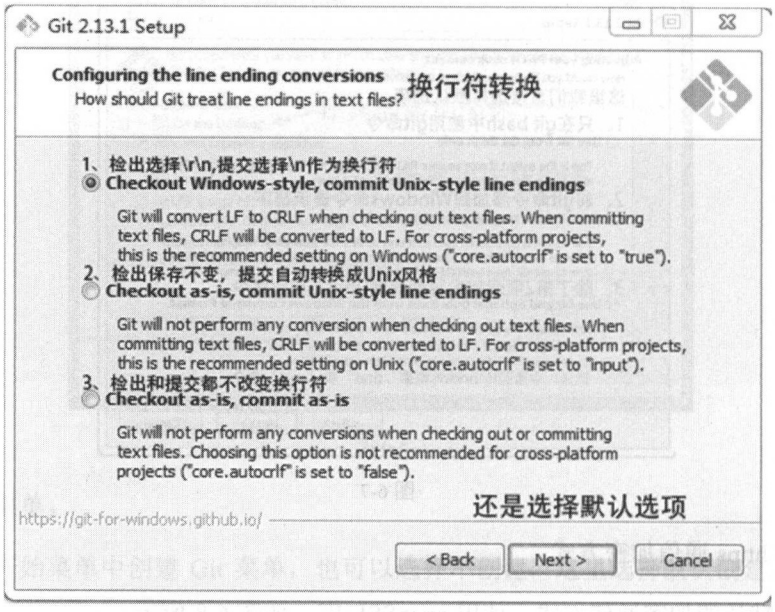


图 6-9

注意：Windows 系统使用回车符（“\r”）和换行符（“\n”）作为行结束标记，而 UNIX 系统则仅使用换行符（“\n”）作为行结束标记。所以在 Windows 和 Linux 系统间传输文件的时候需要确保换行符被准确转换，否则文件内容会乱成一团。

（10）选择终端模式。

使用默认终端模式，MinTTY 是一个 Cygwin 和 MSYS 的虚拟终端，如图 6-10 所示。

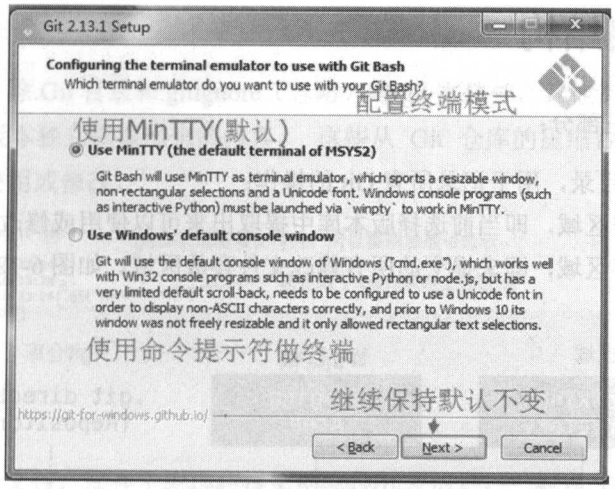


图 6-10

（11）扩展选项。

默认扩展选择，配置完成，直接安装 Git，如图 6-11 所示。

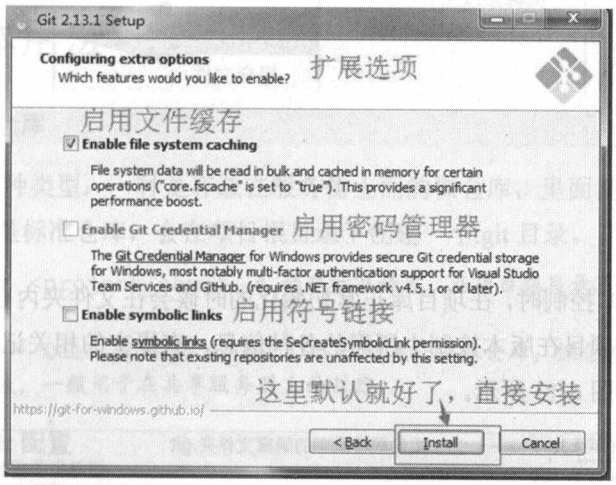


图 6-11

2. Linux 下安装 Git

Ubuntu16.04 默认安装了 Git，所以不需要额外安装，直接使用即可。CentOS 系统可以通过 yum 包管理安装 Git 应用。

```
sudo yum install -y git  # 通过 yum 包管理器安装
```

6.4 Git 项目结构

Git 仓库分成了三部分：

◎ 一个是.git 目录，用于记录所有 Git 的操作；

◎ 一个是工作区域，即当前选择版本库中提取出来可以使用或修改的文件；

◎ 一个是暂存区域，即未提交的所有修改文件存放区域。如图 6-12 所示。

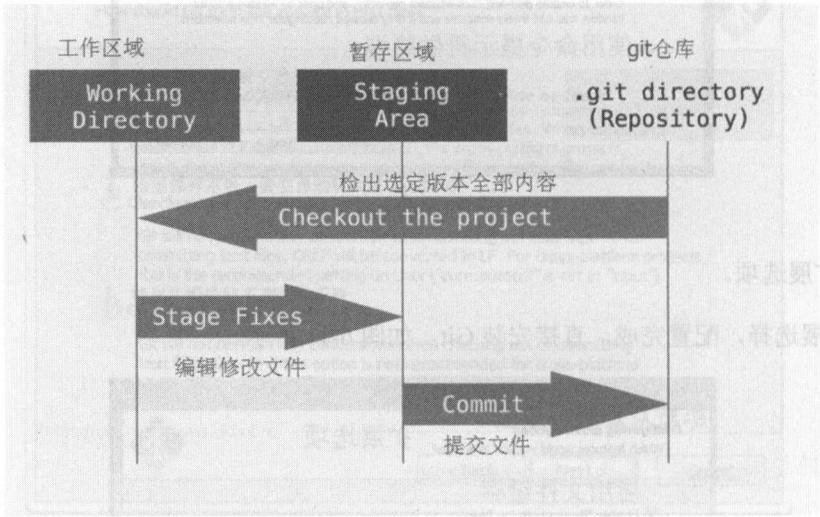


图 6-12

1. .git 目录

使用 Git 进行版本控制时，在项目库正常初始化的时候会在文件夹内自动创建一个.git 的目录。这个目录包含了项目在版本控制中所需的各种信息，有提交的相关记录、提交历史日志、项目配置信息等，如图 6-13 所示。

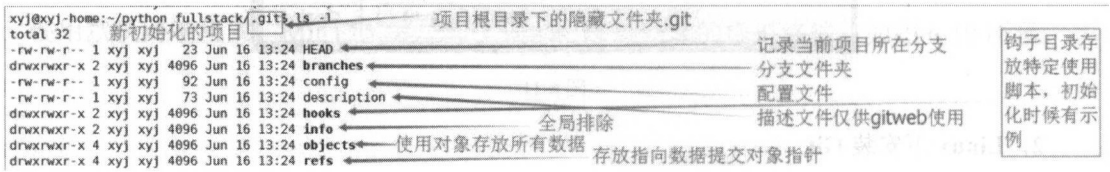


图 6-13

2. 工作目录

在项目文件内，除.Git 目录和.gitignore（忽略）等 git 文件外，其他都是工作目录。工作目录是对项目的某个版本独立提取出来的内容。 这些从 Git 仓库的压缩数据库中提取出来的文件，放在磁盘上供使用或修改。[1]

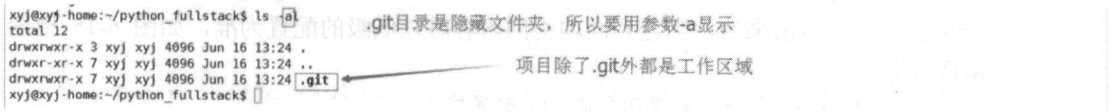


图 6-14

3. 暂存区域

暂存区域是一个文件，保存了下次将提交的文件列表信息，一般在 Git 仓库目录中。暂存区域有时候也被称作“索引”，不过一般说法还是叫暂存区域。[2]我们当前 checkout（检出分支）的文件存会存放在工作目录下，只要一修改工作目录中的文件，修改信息就会记录在.git/index 文件中。

6.5 Git 基本用法

1. 初始化 Git 仓库

Git 仓库分为两种类型：一种是存放在服务器上面的裸仓库，里面没有保存文件，只是存放.git 的内容；一种是标准仓库，会在项目根目录下创建一个.git 目录。

```
    $ git init <project_name>  # 创建标准仓库，在项目根目录下创建一个隐藏的.git
# 文件夹
    $ git init --bare <project_name>  # 创建一个裸仓库，裸仓库只有.git 目录内容，
# 而没有工作区域，一般用于在共享服务器上面创建。
```

2. 查看当前 Git 配置

Git 配置信息分成三个级别，分别存放在三个不同的地方。

1　工作目录解释：
https://git-scm.com/book/zh/v2/%E8%B5%B7%E6%AD%A5-Git-%E5%9F%BA%E7%A1%80

2　暂存区域解释：
https://git-scm.com/book/zh/v2/%E8%B5%B7%E6%AD%A5-Git-%E5%9F%BA%E7%A1%80

◎ 一个是系统级别的配置文件，系统基本配置文件存放在 Git 的安装目录中。

◎ 一个是用户级别配置文件，用户级别配置文件存放在当前用户目录下的.gitconfig 文件内。

◎ 一个是项目级别配置文件，项目级别的配置文件会存放在.git 目录的 config 文件中。

使用 git config --list 显示的 Git 配置信息，是从系统级配置→用户级配置→项目级配置一层层叠加显示出来的，当遇到同项不同内容时以低级的配置为准，如图 6-15 至图 6-17 所示。

```
$ git config --list  # 显示当前 Git 配置信息
$ git config --system --list  # 显示系统级别 Git 配置信息
$ cat .git/config  # 显示项目配置文件
$ cat ~/.gitconfig  # 显示用户级别配置信息
```

```
xyj@xyj-home:~/python_fullstack/.git$ cat config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
xyj@xyj-home:~/python_fullstack/.git$
```

项目级别配置，配置文件存放在.git/config中

图 6-15

```
xyj@xyj-home:~/python_fullstack$ cat ~/.gitconfig
[user]
        name = xyj
        email = xieyingjun@vip.qq.com
xyj@xyj-home:~/python_fullstack$
```

用户级别配置文件存放在当前home对应用户目录下的.gitconfig文件中

图 6-16

```
xyj@xyj-home:~$ git config --system --list
fatal: unable to read config file '/etc/gitconfig': No such file or directory
```

因为我们没有配置系统级配置，所以这个文件没生成

图 6-17

3. 配置当前用户名和邮箱

前面我们说过，用 Git 进行版本控制与集中式版本控制不同，集中版本控制需要验证用户信息后才能提交代码，这样可以识别出谁提交了代码；而分布式版本控制的所有文件都保存在本地磁盘中，当提交代码的时候，需要配置一个用户信息才能被 Git 执行，在团体合作开发的时候用于识别文件是谁提交的，但这个识别并没有验证用户的真伪，如图 6-18 所示。

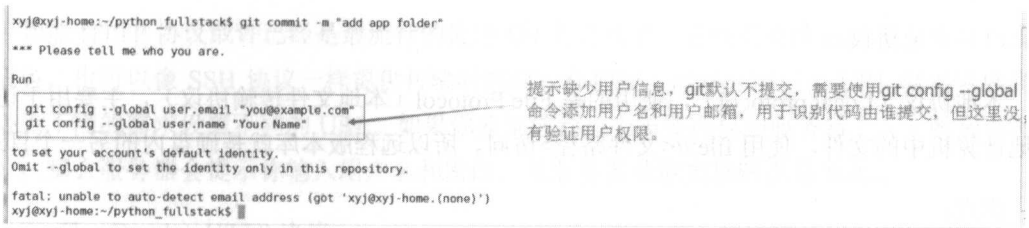


图 6-18

```
$ git config --global user.name "用户名"  # 在用户级配置上设置用户名
$ git config --global user.email "用户邮箱"  # 在用户级配置上设置邮箱
```

如图 6-19 所示。

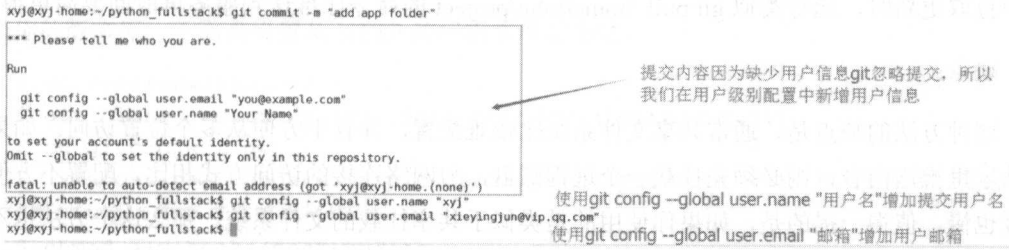


图 6-19

注意：在用户级别配置上设置用户名和邮箱信息，应避免如下情形，假设开发用的电脑为多人使用，并且有一个用户忘记给项目设置用户信息，这时 Git 会把用户信息默认设置为系统级别的信息，而不给出任何提示。

4. 克隆仓库

克隆仓库是从远程服务器上拉取一个完整的仓库到本地磁盘，这样做的好处在于每个人都有一个完整的代码库，避免把鸡蛋放在同一个篮子里。但相对的，每个人都有一个完整仓库，对代码的防泄露又是一个问题。

```
$ git clone <url>  # 从一个远程 Git 仓库中克隆到本地磁盘
```

注意：Git 支持 URL 传输协议：本地协议（Local）、HTTP 协议、SSH（Secure Shell）协议、FTP 协议、Git 协议。[1]

1　协议优缺点对比解释：
https://www.git-scm.com/book/zh/v2/%E6%9C%8D%E5%8A%A1%E5%99%A8%E4%B8%8A%E7%9A%84-Git-%E5%8D%8F%E8%AE%AE，内容略有修改

（1）本地协议。

本地协议（Local protocol），使用的是 File Protocol（本地文件传输协议），主要用于访问本地计算机中的文件，使用 file://<文件路径>访问。所以远程版本库就是硬盘内的另一个目录。

优点：

基于文件系统的版本库的优点是简单，并且直接使用了现有的文件权限和网络访问权限。如果你的团队已经有共享文件系统，那么建立版本库会十分容易。只需像设置其他共享目录一样，把一个裸版本库的副本放到大家都可以访问的路径，并设置好读/写权限就可以了。这也是快速从别人的工作目录中拉取更新的方法。如果你和别人一起合作一个项目，他想让你从版本库中拉取更新时，运行类似 git pull /home/john/project 的命令比推送到服务再取回要简单得多。

缺点：

这种方法的缺点是，通常共享文件系统比较难配置，并且不方便从多个位置访问。如果你想从家里推送内容，则必须先挂载一个远程磁盘，与网络连接的访问方式相比，配置不方便，速度也慢。值得一提的是，如果你使用的是类似于共享挂载的文件系统，那么这个方法也不一定是最快的。访问本地版本库的速度与访问数据的速度是一样的。在同一个服务器上，如果允许 Git 访问本地硬盘，则一般来说，通过 NFS 访问版本库的速度要慢于通过 SSH 访问。

这个协议并不能使仓库避免意外的损坏。每一个用户都有“远程”目录的完整 shell 权限，我们无法阻止他们修改或删除 Git 内部文件或损坏仓库。

（2）HTTP 协议。

Git 通过 HTTP 通信有两种模式。在 Git 1.6.6 版本之前只有一个方式可用，十分简单并且通常是只读模式的。Git 1.6.6 版本引入了一种新的更智能的协议，让 Git 可以像通过 SSH 那样智能地协商和传输数据。之后几年，这个新的 HTTP 协议因为其简单、智能变得十分流行。新版本的 HTTP 协议一般被称为“智能”HTTP 协议，旧版本的一般被称为“哑”HTTP 协议。我们先了解一下“智能”HTTP 协议。

① 智能（Smart）HTTP 协议。

智能 HTTP 协议的运行方式和 SSH 协议及 Git 协议类似，只是运行在标准的 HTTP/S 端口上，并且可以使用各种 HTTP 验证机制，这意味着使用起来要比 SSH 协议简单得多。比如可以使用 HTTP 协议的用户名 / 密码的基础授权，免去设置 SSH 公钥。

智能 HTTP 协议或许已经是最流行的使用 Git 的方式了，它既支持像 git://协议一样设置匿名服务，也可以像 SSH 协议一样提供传输时的授权和加密。而且只用一个 URL 就可以做到，不必为不同的需求设置不同的 URL。如果你要推送到一个需要授权的服务器上（一般来讲都需要），那么服务器会提示你输入用户名和密码。从服务器获取数据时也是如此。

② 哑（Dumb）HTTP 协议。

如果服务器没有提供智能 HTTP 协议的服务，则 Git 客户端会尝试使用更简单的哑 HTTP 协议。在哑 HTTP 协议里，Web 服务器仅把裸版本库当作普通文件来对待，提供文件服务。哑 HTTP 协议的优美之处在于设置起来简单。基本只需把一个裸版本库放在 HTTP 根目录上，设置一个叫作 post-update 的挂钩就可以了。此时，只要能访问 Web 服务器上你的版本库，就可以克隆你的版本库。下面是设置从 HTTP 访问版本库的方法。

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update  # 将示例脚本重命名，需要去
#掉.sample 脚本才能被识别运行
$ chmod a+x hooks/post-update
```

这样就可以了。Git 自带的 post-update 挂钩会默认执行合适的命令（git update-server-info），来确保通过 HTTP 的获取和克隆操作正常工作。这条命令会在你通过 SSH 向版本库推送之后被执行，然后别人就可以通过类似下面的命令来克隆了：

```
$ git clone https://example.com/gitproject.git
```

这里我们使用了 Apache 里设置时常用的路径 /var/www/htdocs，不过你可以使用任何静态 Web 服务器——只需把裸版本库放到正确的目录下即可。Git 的数据是以基本的静态文件形式提供的。通常会在可以提供读 / 写的智能 HTTP 服务和简单的只读的哑 HTTP 服务之间选一个。极少会将二者混合起来提供服务。

优点：

我们将只关注智能 HTTP 协议的优点。

不同的访问方式只需一个 URL，且服务器只需在授权时提示输入授权信息，这两个简便性让终端用户使用 Git 变得非常简单。相比 SSH 协议，可以使用用户名 / 密码授权是一个很大的优势，这样用户就不必在使用 Git 之前先在本地生成 SSH 密钥对再把公钥上传到服务器。对非

资深的使用者，或者系统上缺少 SSH 相关程序的使用者而言，HTTP 协议的可用性是主要的优势。与 SSH 协议类似，HTTP 协议也非常快速和高效。

你也可以在 HTTPS 协议上提供只读版本库的服务，这样你在传输数据的时候就可以加密数据；或者，你甚至可以让客户端使用指定的 SSL 证书。

另一个好处是 HTTPS 协议被广泛使用，一般的企业防火墙都允许这些端口的数据通过。

缺点：

在一些服务器上，架设 HTTPS 协议的服务端会比架设 SSH 协议的服务端棘手一些。除了这一点，用其他协议提供 Git 服务与智能 HTTP 协议相比就几乎没有优势了。

如果你在 HTTP 上使用需授权的推送，那么管理凭证会比使用 SSH 密钥认证麻烦一些。然而，你可以使用凭证存储工具，比如 OSX 的 Keychain 或者 Windows 的凭证管理器。

（3）SSH 协议。

架设 Git 服务器时常用 SSH 协议作为传输协议。因为大多数环境下已经支持通过 SSH 访问——即使没有也比较容易架设。SSH 协议是一个验证授权的网络协议，并且，因为其普遍性，架设和使用都很容易。

通过 SSH 协议克隆版本库，你可以指定一个 ssh://的 URL：

```
$ git clone ssh://user@server/project.git
```

或者使用一个简短的 scp 式的写法：

```
$ git clone user@server:project.git
```

也可以不指定用户，Git 会使用当前登录的用户名。

优点：

用 SSH 协议的优势很多。首先，SSH 架设相对简单——SSH 守护进程很常见，多数管理员都有使用经验，并且多数操作系统都包含了它及相关的管理工具。其次，通过 SSH 访问是安全的——所有传输数据都要经过授权和加密。最后，与 HTTP/S 协议、Git 协议及本地协议一样，SSH 协议很高效，在传输前也会尽量压缩数据。

缺点：

SSH 协议的缺点在于你不能通过它实现匿名访问。即便只是读取数据，使用者也要有通过 SSH 访问你的主机的权限，这使得 SSH 协议不利于开源的项目。如果只在公司网络使用，那么 SSH 协议可能是你唯一要用到的协议。如果要同时提供匿名只读访问和 SSH 协议，那么除了为自己推送架设 SSH 服务外，还要架设一个可以让其他人访问的服务。

（4）Git 协议。

接下来是 Git 协议。这是包含在 Git 里的一个特殊的守护进程；它监听在一个特定的端口（9418），类似于 SSH 服务，但是访问无须任何授权。要想让版本库支持 Git 协议，则需要先创建一个 git-daemon-export-ok 文件——它是 Git 协议守护进程为这个版本库提供服务的必要条件——但是除此之外，没有任何安全措施。要么谁都可以克隆这个版本库，要么谁都不能。这意味着，通常不能通过 Git 协议推送。由于没有授权机制，一旦你开放推送操作，就意味着网络上知道这个项目 URL 的人都可以向项目推送数据。不用说，极少会有人这么做。

优点：

目前，Git 协议是 Git 使用的网络传输协议里速度最快的。如果你的项目有很大的访问量，或者你的项目很庞大并且不需要为写进行用户授权，那么架设 Git 守护进程来提供服务是不错的选择。它使用与 SSH 相同的数据传输机制，但是省去了加密和授权的开销。

缺点：

Git 协议的缺点是缺乏授权机制。把 Git 协议作为访问项目版本库的唯一手段是不可取的。一般的做法是，同时提供 SSH 或者 HTTPS 协议的访问服务，只让少数几个开发者有推送（写）权限，其他人通过 git://访问只有读权限。Git 协议许也是最难架设的。它要求有自己的守护进程，这就要配置 xinetd 或者其他程序，这些工作并不简单。它还要求防火墙开放 9418 端口，但是企业防火墙一般不会开放这个非标准端口。而大型的企业防火墙通常会封锁这个端口。

说明：clone 和 checkout 的区别如下。

git clone 命令是将版本库完整克隆到本地新目录中，在创建好本地库后会自动检出当前活动分支或初始化分支。

git checkout 命令是创建分支或切换分支，使用该命令后会自动更新 HEAD 文件，将其改写成当前分支。

5. 查看文件状态

使用 git status 可以查看当前工作区域内文件的状态，没被跟踪内容会在 Untracked files 中显示，可以通过 git add <file_name>添加被跟踪，如图 6-20 所示。

```
$ git status
```

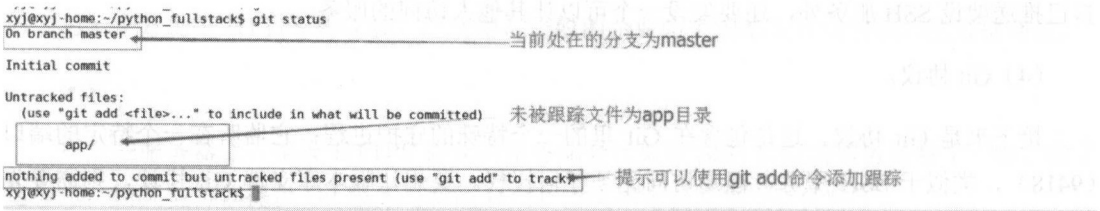


图 6-20

6. 添加文件追踪

使用 git add <file_name>命令将文件添加到 index（索引）文件中，这些文件列表将在下一次提交时记录到仓库，如图 6-21 所示。

```
$ git add app/  # 将 app 目录添加到 index 文件中
```

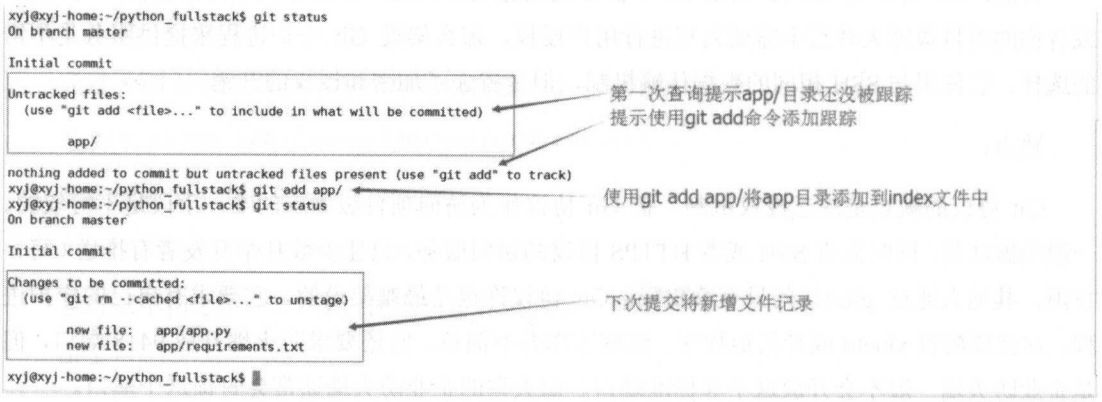


图 6-21

7. 提交代码

使用 git commit 命令将 index 文件中的更改记录提交到本地版本库。使用参数-m 可以直接将要添加的备注信息写入，如图 6-22 所示。

```
$ git commit -m "add app folder"  # 提交到版本库，并写入提交信息
```

```
xyj@xyj-home:~/python_fullstack$ git commit -m "add app folder"
[master (root-commit) e4b93fe] add app folder
 2 files changed, 27 insertions(+)
 create mode 100644 app/app.py
 create mode 100644 app/requirements.txt
xyj@xyj-home:~/python_fullstack$
```

添加好用户信息后，提交
使用参数-m后面msg直接将提交备注写入

图 6-22

8. 查看远程仓库

使用 git remote 命令可以显示当前版本库的远程仓库服务器信息。参数-v 显示远程仓库简写名称和 URL 地址，如图 6-23 所示。

```
$ git remote -v  # 显示版本库连接的远程仓库和 URL
```

```
xyj@xyj-home:~/python_fullstack$ git remote -v
xyj@xyj-home:~/python_fullstack$
```

项目没加远程仓库，所以没显示

使用git remote命令可以显示使用的远程仓库，参数-v显示远程仓库简写和URL

图 6-23

9. 添加远程仓库

使用 git remote add <远程仓库别名> <URL>为本地版本库添加一个远程仓库，如图 6-24 所示。

```
$ git remote add origin
https://github.com/lanmaokafei/python_fullstack.git  # 添加一个远程仓库
```

```
xyj@xyj-home:~/python_fullstack$ git remote -v
xyj@xyj-home:~/python_fullstack$ git remote add origin https://github.com/lanmaokafei/python_fullstack.git
xyj@xyj-home:~/python_fullstack$ git remote -v
origin  https://github.com/lanmaokafei/python_fullstack.git (fetch)
origin  https://github.com/lanmaokafei/python_fullstack.git (push)
xyj@xyj-home:~/python_fullstack$
```

添加一个远程仓库URL

命令格式为：git remote add <远程仓库别名> <URL>

再次使用git remote -v 查询显示出当前版本库连接的远程仓库信息

图 6-24

10. 推送代码

使用 git push <远程仓库别名> <本地分支>将本地版本库推送到远程仓库，如图 6-25 所示。

```
$ git push origin master  # 将本地 master 分支提交到别名为 origin 的远程仓库
```

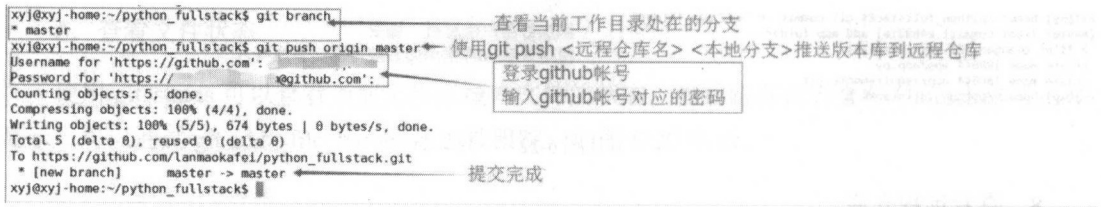


图 6-25

11. 从远程仓库更新代码到本地

将代码推送到远程仓库后，其他非最新版本的用户需要更新最新代码，可以使用 git fetch 或 git pull 命令来更新。区别在于 fetch 获取最新的代码到本地仓库，但不会自动合并分支，需要比较分支差异后合并，而 pull 则直接将远程仓库的版本合并到本地 master 上，所以 git fetch 相对安全些，如图 6-26 和图 6-27 所示。

```
$ git fetch origin master  # 下载 origin 最新的代码
```

图 6-26

```
$ git merge origin/master  # 将 origin/master 分支合并到本地 master 中
```

```
xyj@xyj-OptiPlex-9020:~/new_workspace/python_fullstack$ git merge origin/master
Updating 50f6a3b..9f81d30
Fast-forward
 "doc/00\344\272\244\347\250\277.docx"                                                                | Bin 12413113 -> 14064679 bytes
 doc/sql.sql                                                                                          |  66 ++++++++++++++++
 "doc/~$00\344\272\244\347\250\277.docx"                                                              | Bin 162 -> 0 bytes
 "doc/\347\254\254\344\272\214\350\214\203\345\274\217.mwb"                                           | Bin 10924 -> 11221 bytes
 "doc/\347\254\254\344\272\214\350\214\203\345\274\217.mwb.bak"                                       | Bin 0 -> 10798 bytes
 "jpg/02.\347\216\257\345\242\203\346\220\255\345\273\272/MySQL/\345\217\267\347\247\260.jpg"         | Bin 0 -> 26432 bytes
 .../MySQL/\346\213\211\345\217\226mysql\351\225\234\345\203\217.jpg"                                 | Bin 0 -> 45571 bytes
 .../PostgreSQL/\345\217\267\347\247\260.jpg"                                                         | Bin 0 -> 23893 bytes
 .../PostgreSQL/\346\213\211\345\217\226\345\256\230\346\226\271\351\225\234\345\203\217.jpg"        | Bin 0 -> 55533 bytes
 .../PostgreSQL/\346\213\211\345\217\226\345\256\230\346\226\271\351\225\234\345\203\2172.jpg"       | Bin 0 -> 52021 bytes
 .../docker-gitlab/gitlab\351\235\242\346\235\277.jpg"                                                | Bin 0 -> 89095 bytes
 .../docker-gitlab/\345\210\233\345\273\272\346\226\260\351\241\271\347\233\256.jpg"                  | Bin 0 -> 96227 bytes
 .../docker-gitlab/\346\263\250\345\206\214gitlab\347\224\250\346\210\267.jpg"                        | Bin 0 -> 87162 bytes
 .../docker-gitlab/\350\256\276\347\275\256\344\273\223\345\272\223.jpg"                              | Bin 0 -> 113896 bytes
 .../docker-gitlab/\351\241\271\347\233\256\351\235\242\346\235\277.jpg"                              | Bin 0 -> 99706 bytes
 ...6\346\267\273\345\212\240\345\210\260\347\216\257\345\242\203\345\217\230\351\207\217.jpg"       | Bin 0 -> 97058 bytes
 .../docker/Docker-Compose\344\270\213\350\275\275\351\241\265\351\235\242.jpg"                       | Bin 0 -> 100908 bytes
 .../docker/Docker-remote-api\347\273\221\345\256\232\347\253\257\345\217\243.jpg"                    | Bin 0 -> 93985 bytes
 .../docker/Docker-remote-api\347\273\221\345\256\232\347\253\257\345\217\2432.jpg"                   | Bin 0 -> 95822 bytes
 ...r\351\225\234\345\203\217\345\233\275\345\206\205\346\272\220\345\212\240\351\200\237.jpg"       | Bin 0 -> 25523 bytes
 ...7\347\224\250\350\205\276\350\256\257\344\272\221\345\212\240\351\200\237\346\272\220.jpg"       | Bin 0 -> 66752 bytes
 ...\347\224\250\350\205\276\350\256\257\344\272\221\345\212\240\351\200\237\346\272\2202.jpg"       | Bin 0 -> 70071 bytes
 ...\347\224\250\350\205\276\350\256\257\344\272\221\345\212\240\351\200\237\346\272\2203.jpg"       | Bin 0 -> 26781 bytes
 .../docker/\345\201\234\346\255\242\345\256\271\345\231\250.jpg"                                     | Bin 0 -> 122045 bytes
 ...7\345\207\272\350\277\220\350\241\214\345\256\271\345\231\250\344\277\241\346\201\257.jpg"       | Bin 0 -> 103204 bytes
 .../docker/\345\220\257\345\212\250\345\256\271\345\231\250.jpg"                                     | Bin 0 -> 105527 bytes
```

将origin/master分支合并到当前master

图 6-27

12. 版本标记

很多书或网上会把版本标记翻译成里程碑，即给当前提交定义一个标签，标记出当前提交内容有别于其他提交。例如，在开发完一个新功能后，可以将其标记一个 v1.1，代表这个功能开发完成，方便以后历史中定位这次提交。

Git 有两种标签类型：一种是 lightweight 轻量级标签，只有版本号无其他信息；另一种是 annotated 附注标签，标签附带一条信息，可以让别人快速识别标签作用，建议使用这种标签。

使用 git tag -n[数字] 显示当前分支下的标签信息，参数-n 代表显示备注信息行数，如图 6-28 所示。

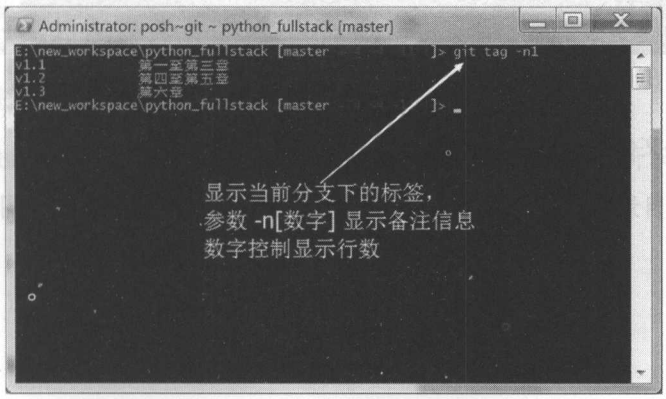


图 6-28

使用 git tag -a <版本号> -m “备注” 为最新提交打上标签，如图 6-29 所示。

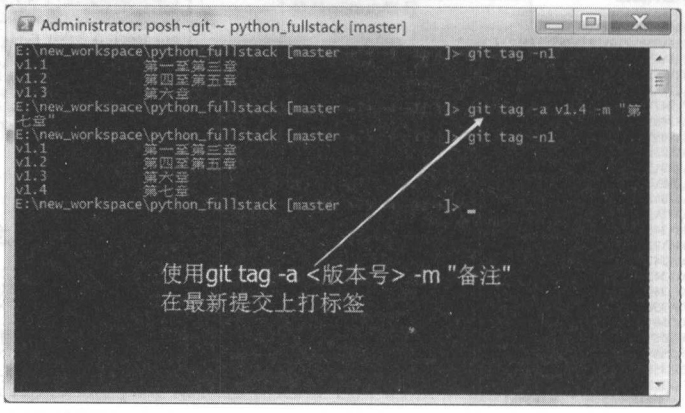


图 6-29

使用 git show <版本号>显示对应标签的版本信息和提交差异，如图 6-30 所示。

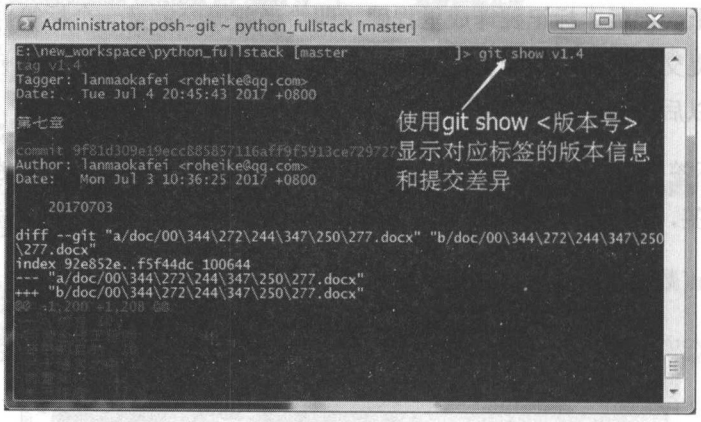


图 6-30

13. 添加忽略文件

在当前项目根目录下创建一个.gitignore 文件，用于保存忽略列表，如图 6-3 所示。

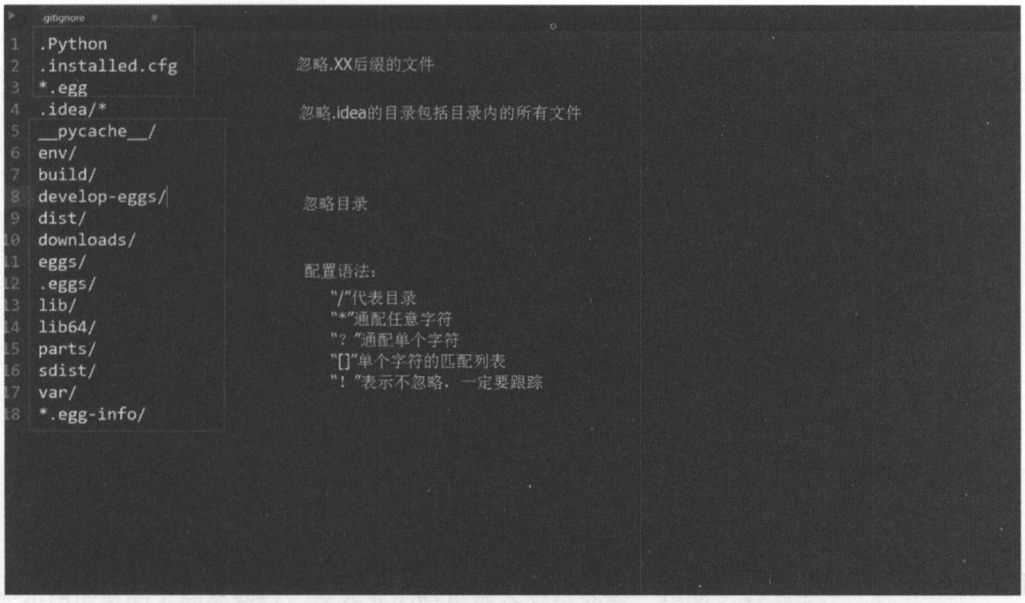


图 6-31

配置语法如下：

“/”代表目录；

“*”代表通配任意字符；

“？”代表通配单个字符；

“[]”代表单个字符的匹配列表；

“！”表示不忽略，一定要跟踪。

14. 查看当前处在的分支

使用命令 git branch 可以查看当前工作目录所在的分支，如图 6-32 所示。

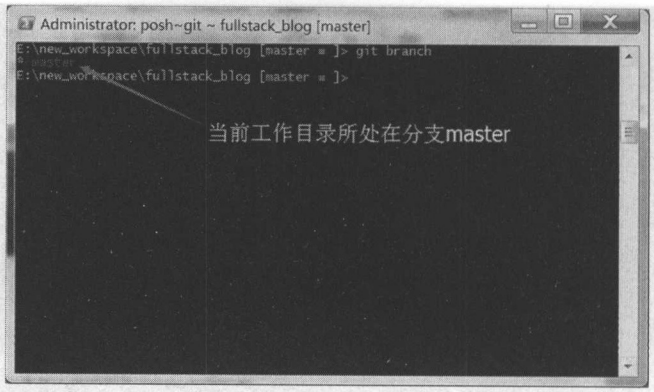


图 6-32

15. 创建分支

在家里开发使用的是 PostgreSQL 数据库，到公司后没有在线的数据就切换到 SQLite，这样我们就创建一个新的分支以便在公司开发。使用命令 git checkout -b jsb 创建并切换到 jsb 分支，命令 git checkout -b 等同于同时执行了命令 git branch jsb 创建分支和命令 git checkout jsb 切换到 jsb 分支，如图 6-33 所示。

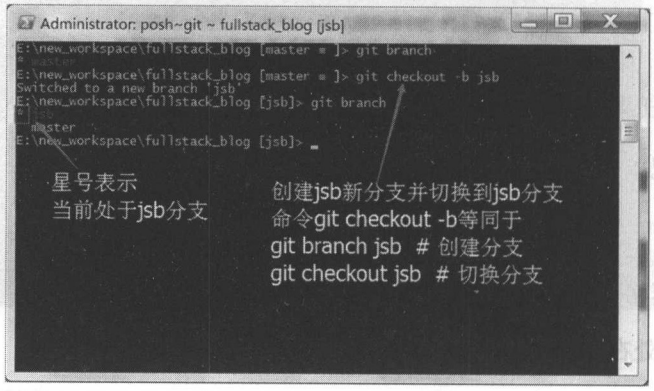


图 6-33

16. 合并分支

当回到家时再把在公司开发的代码和家里的版本库合并分支，切换回 master 分支，使用命令 git merge<branch_name>合并两个分支，如图 6-34 所示。

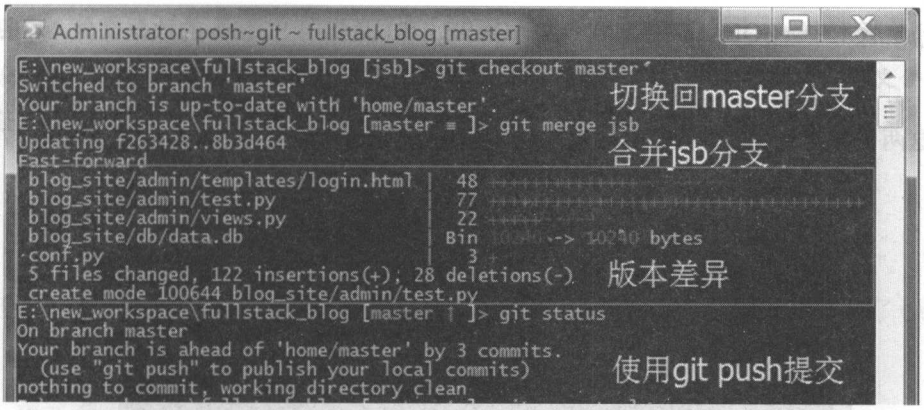


图 6-34

17. 解决冲突

之前使用了不同的忽略语句，两个分支间没有冲突，但是如果两个分支同时修改了同一个文件相同位置的不同参数时，在合并的时候就会产生冲突，如图 6-35 所示。

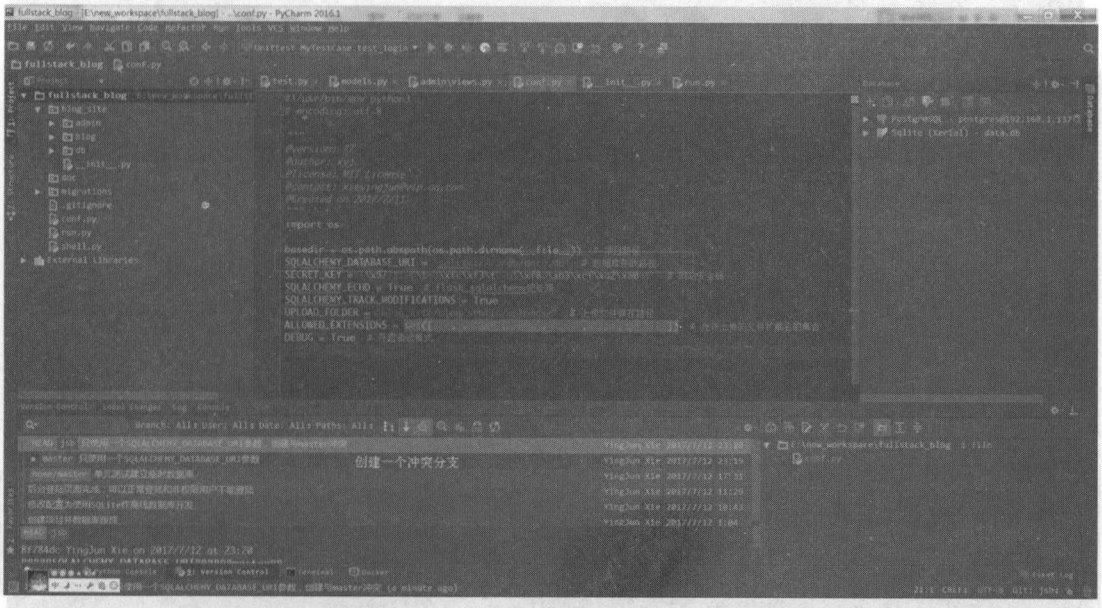


图 6-35

在 master 版本中，SQLALCHEMY_DATABASE_URI 是连接 PostgreSQL 的数据连接 'postgresql://xyj:lanmaokafei@192.168.1.137/postgres'。

在 jsb 版本中，SQLALCHEMY_DATABASE_URI 是指向 SQLite 的数据文件 'sqlite:///./db/data.db' 。

当这两个分支合并的时候就会产生冲突，需要人工修改才能合并，如图 6-36 和图 6-37 所示。

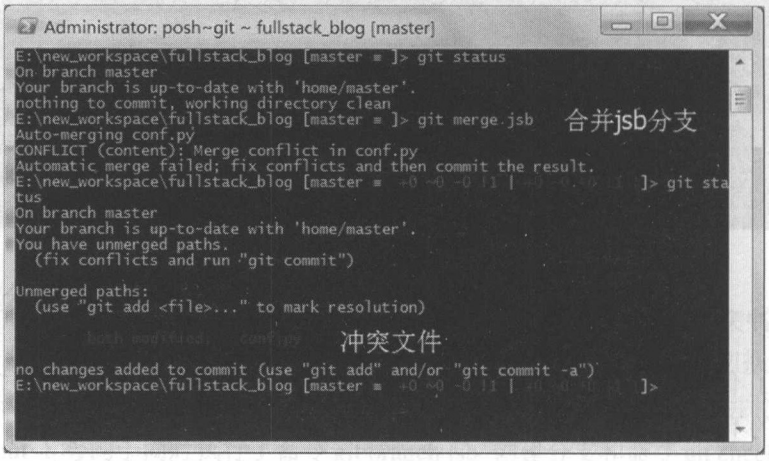


图 6-36

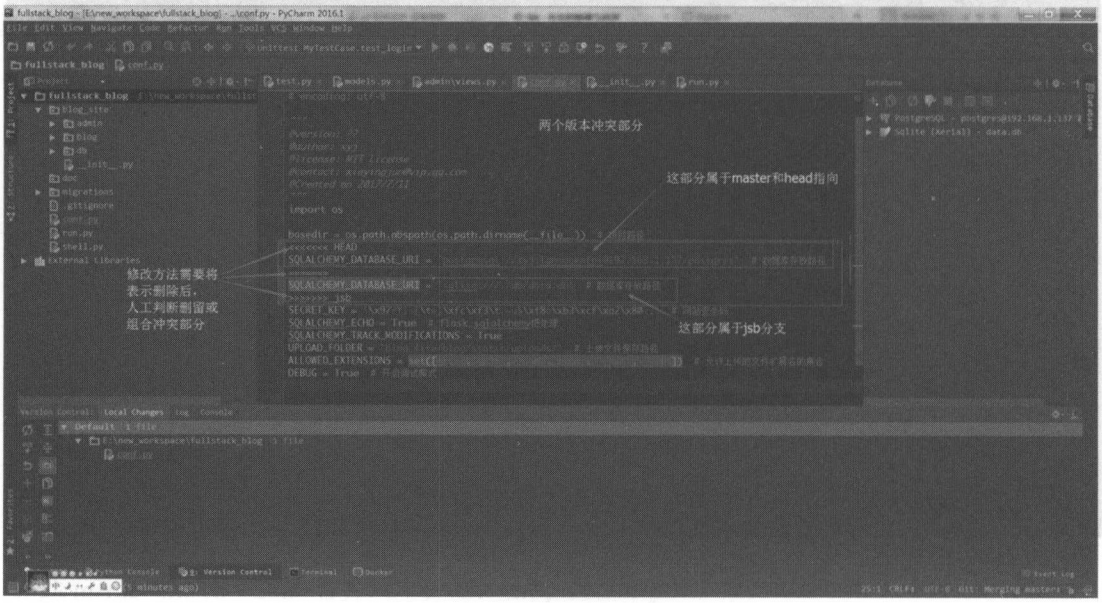


图 6-37

说明：人工修改冲突部分，需要将自动生成的<<<<<<< HEAD、=======、>>>>>>> jsb 全部删除，自行判断冲突部分需要保留什么内容或者将两者融合，修改完成后保存文件，重新使用命令 git add 添加文件，后再提交一次，即可解决冲突问题。

6.6　CentOS 系统搭建 Git 服务器

由于 Ubuntu16.04 系统直接安装好了 Git，可以省略编译安装过程，所以这里选择了 CentOS 7 来安装，下面演示在 Linux 下安装 Git 的过程。

（1）更新系统所有软件包，如图 6-38 所示。

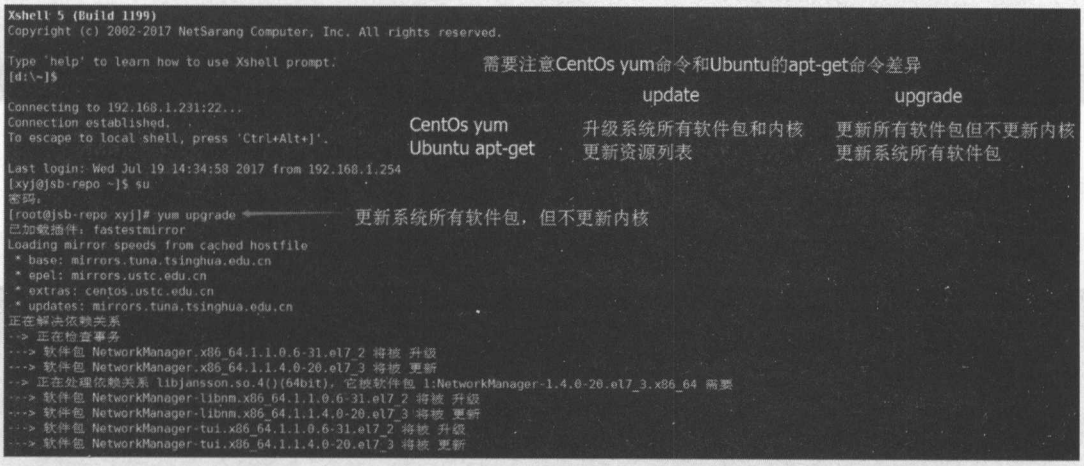


图 6-38

CentOS 系统包管理器命令和 Ubuntu 包管理器命令的差异如表 6-17 所示。

表 6-17

	update	upgrade
CentOS yum	更新所有包和内核	更新所有包但不更新内核
Ubuntu apt-get	更新资源列表	更新所有包

（2）安装依赖库和编译工具。

为了后续安装能正常进行，我们先来安装一些相关依赖库和编译工具，如图 6-39 所示。

```
# yum install -y curl-devel expat-devel gettext-devel openssl-devel
zlib-devel  # 安装 git 所需依赖库
```

```
  systemd-libs.x86_64 0:219-30.el7_3.9                  systemd-sysv.x86_64 0:219-30.el7_3.9
  tar.x86_64 2:1.26-31.el7                              tuned.noarch 0:2.7.1-3.el7_3.2
  tzdata.noarch 0:2017b-1.el7                           util-linux.x86_64 0:2.23.2-33.el7_3.2
  vim-common.x86_64 2:7.4.160-1.el7_3.1                 vim-enhanced.x86_64 2:7.4.160-1.el7_3.1
  vim-filesystem.x86_64 2:7.4.160-1.el7_3.1             vim-minimal.x86_64 2:7.4.160-1.el7_3.1
  virt-what.x86_64 0:1.13-8.el7                         wget.x86_64 0:1.14-13.el7
  wpa_supplicant.x86_64 1:2.0-21.el7_3                  xfsprogs.x86_64 0:4.5.0-10.el7_3
  xz.x86_64 0:5.2.2-1.el7                               xz-libs.x86_64 0:5.2.2-1.el7
  yum.noarch 0:3.4.3-150.el7.centos                     yum-plugin-fastestmirror.noarch 0:1.1.31-40.el7
  zlib.x86_64 0:1.2.7-17.el7

完毕!
[root@jsb-repo xyj]# yum install -y curl-devel expat-devel gettext-devel openssl-devel zlib-devel
已加载插件：fastestmirror
Loading mirror speeds from cached hostfile
 * base: mirrors.tuna.tsinghua.edu.cn
 * epel: mirrors.ustc.edu.cn
 * extras: centos.ustc.edu.cn
 * updates: mirrors.tuna.tsinghua.edu.cn
正在解决依赖关系
--> 正在检查事务
---> 软件包 expat-devel.x86_64.0.2.1.0-10.el7_3 将被 安装
---> 软件包 gettext-devel.x86_64.0.0.18.2.1-4.el7 将被 安装
--> 正在处理依赖关系 gettext-common-devel = 0.18.2.1-4.el7，它被软件包 gettext-devel-0.18.2.1-4.el7.x86_64 需要
--> 正在处理依赖关系 git，它被软件包 gettext-devel-0.18.2.1-4.el7.x86_64 需要
---> 软件包 libcurl-devel.x86_64.0.7.29.0-35.el7.centos 将被 安装
base/7/x86_64/filelists_db                                                    | 6.6 MB  00:00:02
dockerrepo/filelists_db                                                       |  40 kB  00:00:00
epel/x86_64/filelists_db                          12% [======                 ] 165 kB/s | 1.0 MB  00:00:43 ETA
```

图 6-39

安装编译工具，如图 6-40 所示。

```
# yum install -y gcc perl-ExtUtils-MakeMaker  # 安装 gcc
```

```
[root@jsb-repo xyj]# yum install -y gcc perl-ExtUtils-MakeMaker
已加载插件：fastestmirror
Loading mirror speeds from cached hostfile
 * base: mirrors.tuna.tsinghua.edu.cn
 * epel: mirrors.ustc.edu.cn
 * extras: centos.ustc.edu.cn
 * updates: mirrors.tuna.tsinghua.edu.cn
正在解决依赖关系
--> 正在检查事务
---> 软件包 gcc.x86_64.0.4.8.5-11.el7 将被 安装
--> 正在处理依赖关系 cpp = 4.8.5-11.el7，它被软件包 gcc-4.8.5-11.el7.x86_64 需要
--> 正在处理依赖关系 glibc-devel >= 2.2.90-12，它被软件包 gcc-4.8.5-11.el7.x86_64 需要
--> 正在处理依赖关系 libmpfr.so.4()(64bit)，它被软件包 gcc-4.8.5-11.el7.x86_64 需要
--> 正在处理依赖关系 libmpc.so.3()(64bit)，它被软件包 gcc-4.8.5-11.el7.x86_64 需要
---> 软件包 perl-ExtUtils-MakeMaker.noarch.0.6.68-3.el7 将被 安装
--> 正在处理依赖关系 perl(Test::Harness)，它被软件包 perl-ExtUtils-MakeMaker-6.68-3.el7.noarch 需要
--> 正在处理依赖关系 perl(ExtUtils::Packlist)，它被软件包 perl-ExtUtils-MakeMaker-6.68-3.el7.noarch 需要
--> 正在处理依赖关系 perl(ExtUtils::Manifest)，它被软件包 perl-ExtUtils-MakeMaker-6.68-3.el7.noarch 需要
--> 正在处理依赖关系 perl(ExtUtils::Installed)，它被软件包 perl-ExtUtils-MakeMaker-6.68-3.el7.noarch 需要
--> 正在处理依赖关系 perl(ExtUtils::Install)，它被软件包 perl-ExtUtils-MakeMaker-6.68-3.el7.noarch 需要
--> 正在检查事务
---> 软件包 cpp.x86_64.0.4.8.5-11.el7 将被 安装
---> 软件包 glibc-devel.x86_64.0.2.17-157.el7_3.5 将被 安装
--> 正在处理依赖关系 glibc-headers = 2.17-157.el7_3.5，它被软件包 glibc-devel-2.17-157.el7_3.5.x86_64 需要
--> 正在处理依赖关系 glibc-headers，它被软件包 glibc-devel-2.17-157.el7_3.5.x86_64 需要
---> 软件包 libmpc.x86_64.0.1.0.1-3.el7 将被 安装
---> 软件包 mpfr.x86_64.0.3.1.1-4.el7 将被 安装
---> 软件包 perl-ExtUtils-Install.noarch.0.1.58-291.el7 将被 安装
--> 正在处理依赖关系 perl-devel，它被软件包 perl-ExtUtils-Install-1.58-291.el7.noarch 需要
```

图 6-40

（3）下载 Git 源码，如图 6-41 所示。

```
# cd /usr/local/src  #切换至/usr/local/src 目录
```

```
    # wget https://www.kernel.org/pub/software/scm/git/git-2.10.0.tar.gz
# 下载 Git 源码
```

```
  验证中      : glibc-headers-2.17-157.el7_3.5.x86_64                                                    15/17
  验证中      : perl-ExtUtils-MakeMaker-6.68-3.el7.noarch                                                16/17
  验证中      : cpp-4.8.5-11.el7.x86_64                                                                  17/17

已安装:
  gcc.x86_64 0:4.8.5-11.el7                          perl-ExtUtils-MakeMaker.noarch 0:6.68-3.el7

作为依赖被安装:
  cpp.x86_64 0:4.8.5-11.el7                  gdbm-devel.x86_64 0:1.10-8.el7          glibc-devel.x86_64 0:2.17-157.el7_3.5
  glibc-headers.x86_64 0:2.17-157.el7_3.5    kernel-headers.x86_64 0:3.10.0-514.26.2.el7  libdb-devel.x86_64 0:5.3.21-19.el7
  libmpc.x86_64 0:1.0.1-3.el7                mpfr.x86_64 0:3.1.1-4.el7               perl-ExtUtils-Install.noarch 0:1.58-291.el7
  perl-ExtUtils-Manifest.noarch 0:1.61-244.el7  perl-ExtUtils-ParseXS.noarch 1:3.18-2.el7  perl-Test-Harness.noarch 0:3.28-3.el7
  perl-devel.x86_64 4:5.16.3-291.el7         pyparsing.noarch 0:1.5.6-9.el7          systemtap-sdt-devel.x86_64 0:3.0-7.el7

完毕!
[root@jsb-repo xyj]# cd /usr/local/src
[root@jsb-repo src]# wget https://www.kernel.org/pub/software/scm/git/git-2.10.0.tar.gz
--2017-07-19 15:32:15--  https://www.kernel.org/pub/software/scm/git/git-2.10.0.tar.gz
正在解析主机 www.kernel.org (www.kernel.org)... 147.75.110.187, 2604:1380:3000:3500::3
正在连接 www.kernel.org (www.kernel.org)|147.75.110.187|:443... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度：6048363 (5.8M) [application/x-gzip]
正在保存至："git-2.10.0.tar.gz"

100%[=======================================================================================>] 6,048,363   26.7KB/s 用时 5m 10s

2017-07-19 15:37:25 (19.1 KB/s) - 已保存 "git-2.10.0.tar.gz" [6048363/6048363])

[root@jsb-repo src]#
```

图 6-41

（4）解压 Git 源代码，如图 6-42 所示。

```
    # tar -zvxf git-2.10.0.tar.gz  # 将下载的源码解压到当前文件夹
```

```
git-2.10.0/wrap-for-bin.sh
git-2.10.0/wrapper.c
git-2.10.0/write_or_die.c
git-2.10.0/ws.c
git-2.10.0/wt-status.c
git-2.10.0/wt-status.h
git-2.10.0/xdiff-interface.c
git-2.10.0/xdiff-interface.h
git-2.10.0/xdiff/
git-2.10.0/xdiff/xdiff.h
git-2.10.0/xdiff/xdiffi.c
git-2.10.0/xdiff/xdiffi.h
git-2.10.0/xdiff/xemit.c
git-2.10.0/xdiff/xemit.h
git-2.10.0/xdiff/xhistogram.c
git-2.10.0/xdiff/xinclude.h
git-2.10.0/xdiff/xmacros.h
git-2.10.0/xdiff/xmerge.c
git-2.10.0/xdiff/xpatience.c
git-2.10.0/xdiff/xprepare.c
git-2.10.0/xdiff/xprepare.h
git-2.10.0/xdiff/xtypes.h
git-2.10.0/xdiff/xutils.c
git-2.10.0/xdiff/xutils.h
git-2.10.0/zlib.c
git-2.10.0/configure
git-2.10.0/version
git-2.10.0/git-gui/version
[root@jsb-repo src]# tar -zvxf git-2.10.0.tar.gz
```

图 6-42

（5）解压后进入 git-2.10.0 文件夹并执行编译，如图 6-43 所示。

```
    # cd git-2.10.0 && make all prefix=/usr/local/git  # 进入文件夹并编译
```

```
[root@jsb-repo src]# cd git-2.10.0 && make all prefix=/usr/local/git
GIT_VERSION = 2.10.0
    * new build flags
    CC credential-store.o
    * new link flags
    CC common-main.o
    CC abspath.o
    CC advice.o
    CC alias.o
    CC alloc.o
    CC archive.o
    CC archive-tar.o
    CC archive-zip.o
    CC argv-array.o
    * new prefix flags
    CC attr.o
    CC base85.o
    CC bisect.o
    CC blob.o
    CC branch.o
    CC bulk-checkin.o
    CC bundle.o
    CC cache-tree.o
    CC color.o
    CC column.o
    CC combine-diff.o
    CC commit.o
    CC compat/obstack.o
    CC compat/terminal.o
```

图 6-43

（6）编译完成后，安装到 /usr/local/git 目录下，如图 6-44 所示。

```
# make install prefix=/usr/local/git  # 编译完成后使用 make install
#<install_path>命令安装后面接安装路径到/usr/local/git 目录下
```

```
[root@jsb-repo src]# cd git-2.10.0 && make all prefix=/usr/local/git
GIT_VERSION = 2.10.0
    * new build flags
    CC credential-store.o
    * new link flags
    CC common-main.o
    CC abspath.o
    CC advice.o
    CC alias.o
    CC alloc.o
    CC archive.o
    CC archive-tar.o
    CC archive-zip.o
    CC argv-array.o
    * new prefix flags
    CC attr.o
    CC base85.o
    CC bisect.o
    CC blob.o
    CC branch.o
    CC bulk-checkin.o
    CC bundle.o
    CC cache-tree.o
    CC color.o
    CC column.o
    CC combine-diff.o
    CC commit.o
    CC compat/obstack.o
    CC compat/terminal.o
```

图 6-44

（7）创建 Git 账号，如图 6-45 所示。

```
# useradd -m gituser  # 创建一个用户名为 gituser 的用户
# passwd gituser  # 为 gituser 用户设置密码
```

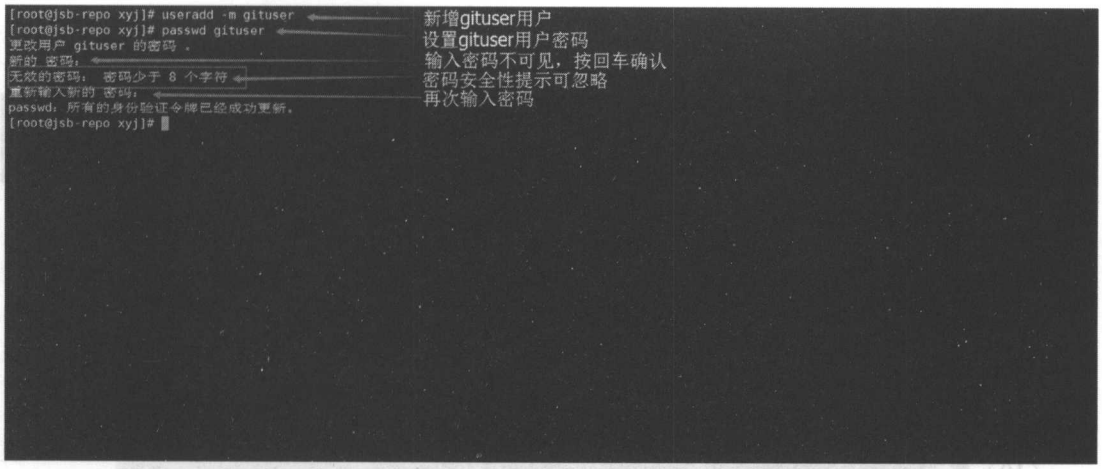


图 6-45

（8）创建 Git 仓库并初始化，如图 6-46 所示。

```
# mkdir -p /data/repository  # 创建目录，用于存放远程仓库
# cd /data/repository/ && git init --bare test.git  # 进入目录并用 git 命
# 令初始化一个 test.git 的裸仓库
```

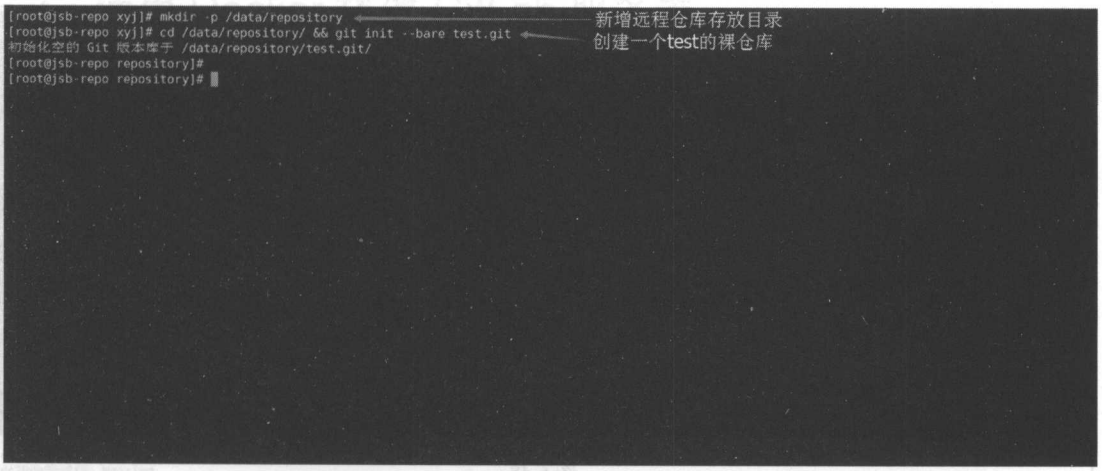


图 6-46

（9）更改仓库目录权限，如图 6-47 所示。

```
# chown -R gituser:gituser /data/repository  # 将/data/repository 目录
# 所有者更改成 gituser 组中的 gituser 用户
```

```
    # chmod 755 /data/repository  # 将目录权限更改成所有者全部权限，组和其他用户
# 拥有读和执行权
```

```
[root@jsb-repo repository]# chown -R gituser:gituser /data/repository
[root@jsb-repo repository]# chmod 755 /data/repository
[root@jsb-repo repository]#
```

更改/data/repository目录所有者

更改/data/repository目录权限

图 6-47

说明：使用 chmod 给目录修改权限，这里 755 的含义如下。

◎ 第一个数字 7 指目录所有者即 gituser 拥有所有权限。

◎ 第二个数字 5 指 gituser 所在的用户组只拥有读和执行权限。

◎ 第三个数字 5 指其他用户有读和执行的权限。

（10）修改 gituser，登录 shell，如图 6-48 所示。

更改/etc/passwd 登录清单文件，将 gituser 的登录命令解释程序由 Git 提供的 shell 完成。

```
    # vim /etc/passwd  # 使用 vim 编辑/etc/passwd 文件
    gituser:x:500:500::/home/gituser:/usr/local/git/bin/git-shell  # 将最
# 后 gituser 的命令解释程序(shell)更改成 git-shell，令 gituser 用户登录后只能执行
# git-shell
```

```
[root@jsb-repo repository]# vim /etc/passwd
```

将gituser用户登录的shell更改成git-shell

图 6-48

（11）克隆远程仓库到本地，如图 6-49 所示。

这里演示的是在同一台服务器上面克隆，而现实场景中，一般是在两台服务器之间克隆，确保远程仓库和工作目录不在同一台服务器上，避免鸡蛋都放在同一个篮子里。

```
> git clone gituser@192.168.1.231:/data/repository/test.git # 使用 git
# clone 命令克隆远程仓库
```

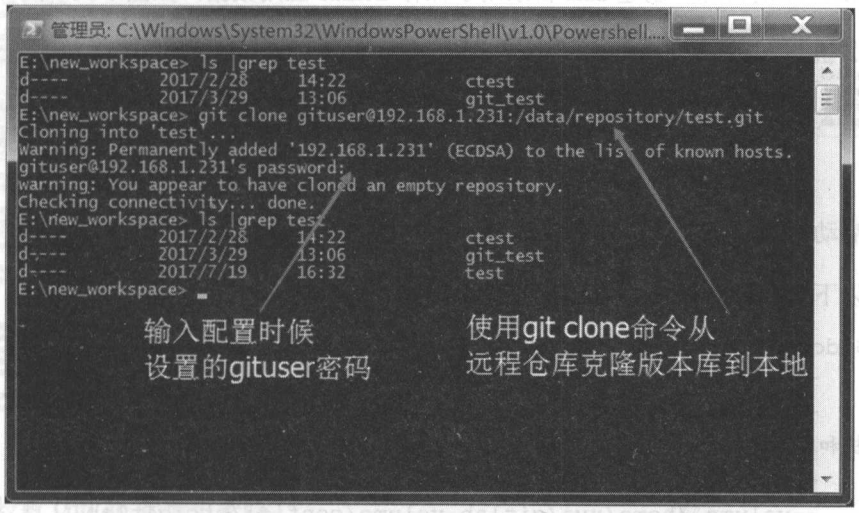


图 6-49

6.7　使用 Docker 搭建 GitLab 服务器

使用 Docker 部署 GitLab 服务器是一件十分简单的事，只需要一条命令就可以把官方编译好的 GitLab 下载到本地运行使用，如图 6-50 所示。

```
$ docker pull gitlab/gitlab-ce  # 从 Docker Hub 服务器上下载由 GitLab 官方
# 提供的 GitLab 社区版镜像
```

图 6-50

（1）持久数据存储位置。

因为容器不能持久保存数据，所以我们需要在用户目录下创建 gitlab_volume 目录，用于容

器映射到本地磁盘保存，在目录内创建 config 目录用于存放 GitLab 的配置，创建 logs 目录用于保存 GitLab 产生的日志，创建 data 目录用于保存 GitLab 应用数据，如图 6-51 所示。

图 6-51

（2）启动 GitLab 容器。

使用以下命令启动 GitLab 容器，如图 6-52 所示。

```
$ docker run --detach \
    --hostname 192.168.1.137 \  # 设置主机 IP 或域名
    --publish 443:443 --publish 8000:8000 --publish 6050:22 \  # 绑定宿
# 主机和容器的端口
    --name gitlab \  # 定义容器别名
    --volume /home/xyj/gitlab_volume/config:/etc/gitlab \  # 将本地磁盘
# 映射到容器内，用于持久保存 GitLab 配置
    --volume /home/xyj/gitlab_volume/logs:/var/log/gitlab \  # 将本地磁
# 盘映射到容器内用于持久保存 GitLab 运行日志
    --volume /home/xyj/gitlab_volume/data:/var/opt/gitlab \  # 将本地磁
# 盘映射到容器内用于持久保存 GitLab 数据
    gitlab/gitlab-ce:latest  # 选择启动镜像
```

图 6-52

（3）修改 GitLab 服务端口。

修改 gitlab.rb 配置文件，修改 GitLab 默认 http 提供服务端口为 8000，如图 6-53 所示。

```
$ sudo docker exec -it gitlab vi /etc/gitlab/gitlab.rb  # 修改容器内
# gitlab.rb 配置
```

```
external_url "http://192.168.1.137:8000"
```

图 6-53

（4）设置 GitLab 管理员密码。

在浏览器打开上一步设定的 hostname，这里是内网 IP。打开 http://192.168.1.137:8000 就可以自动跳转到设置 GitLab 管理员密码页面，如图 6-54 所示。

图 6-54

（5）注册 GitLab 账号。

在登录页面单击 Register，选择注册新用户。

图 6-55

（6）登录 GitLab。

设置好密码后会自动跳转到登录页面，使用 root 账号和上一步设置的密码可以登录 GitLab，如图 6-56 所示。

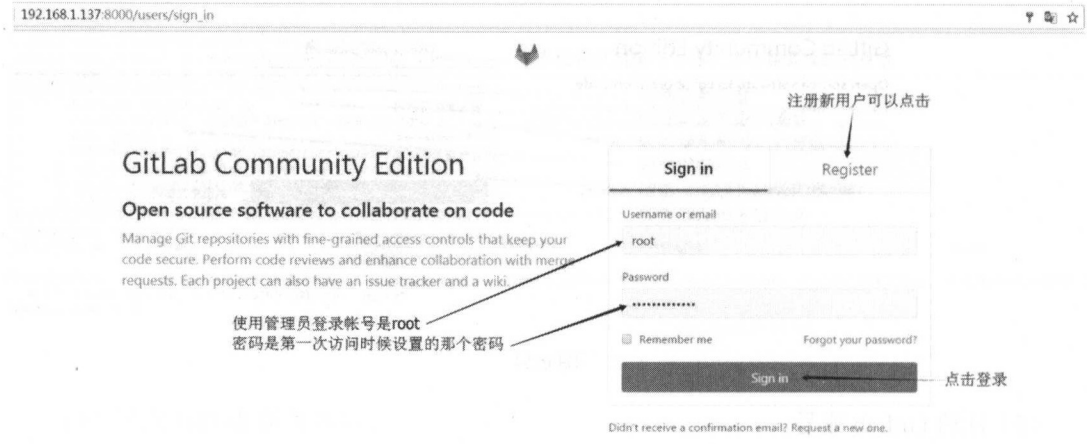


图 6-56

（7）创建新项目。

因为新账户没有任何项目，所以会显示 Gitlab 面板，单击“New project”创建一个新项目，如图 6-57 和图 6-58 所示。

图 6-57

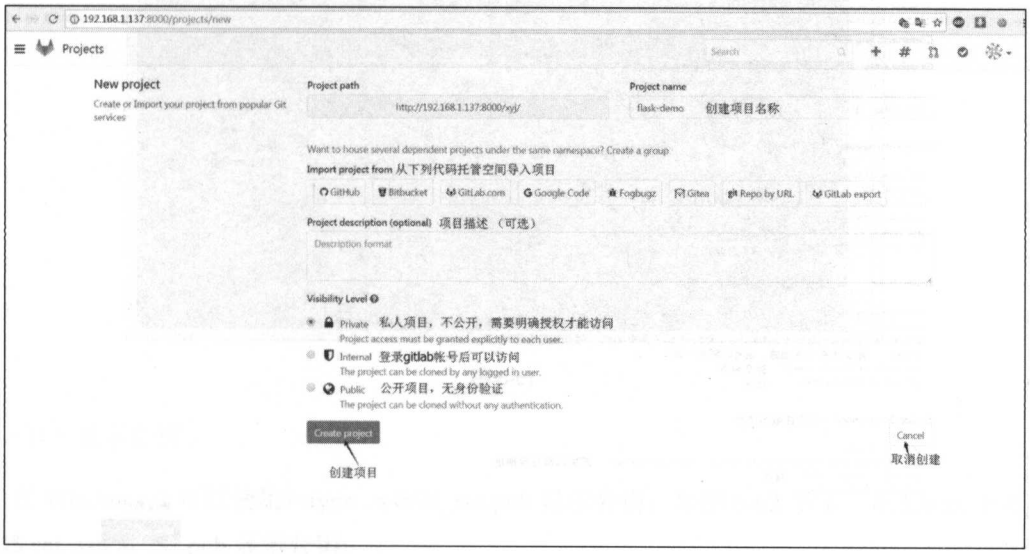


图 6-58

（8）项目面板。

创建好项目仓库后，会提示该仓库还是空的。项目面板如图 6-59 所示。

图 6-59

（9）将本地项目推送到 GitLab 服务器。

使用前面 Git 章节中介绍的添加远程仓库方法，在本地仓库中添加远程仓库地址，或者按图 6-60 所示方法设置仓库。

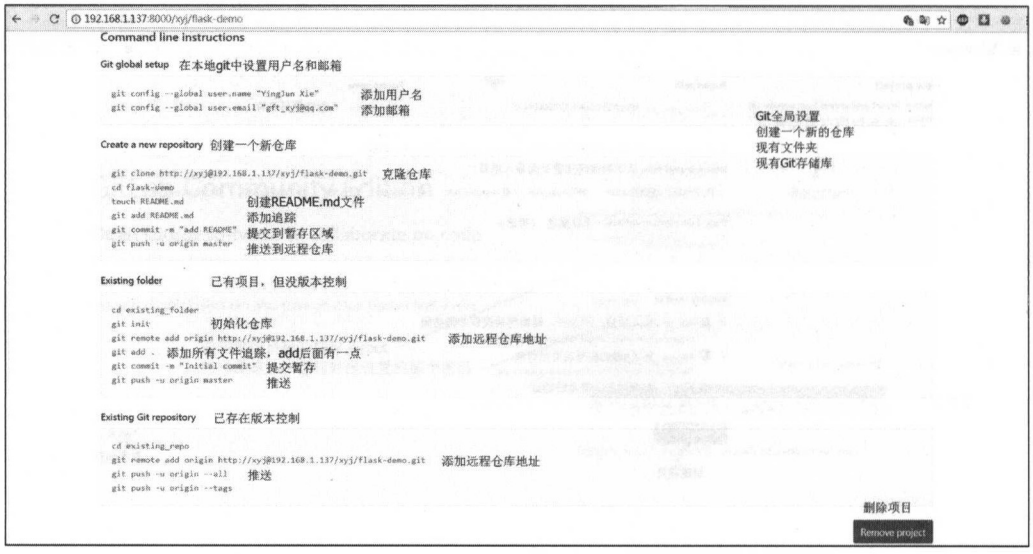


图 6-60

（10）生成 SSH 密钥。

在 Windows 或 Linux 的 git bash 下，可以使用下面的命令生成新 SSH 密钥，如图 6-61 所示。

```
    > ssh-keygen -t rsa -C "xieyingjun@vip.qq.com" -b 4096  # 生成 4096 长度
# 的 RSA 密钥
```

使用 ssh-keygen 生成 SSH 非对称密钥。

参数-t：指定密钥类型为 rsa。

参数-C：注释。

参数-b：指定密钥长度。

图 6-61

（11）显示公钥。

在 Windows 下可以使用> type .\ssh\id_rsa.pub 显示公钥，如图 6-62 所示。在 Linux 下可以使用$ cat .ssh/id_rsa.pub 显示公钥。

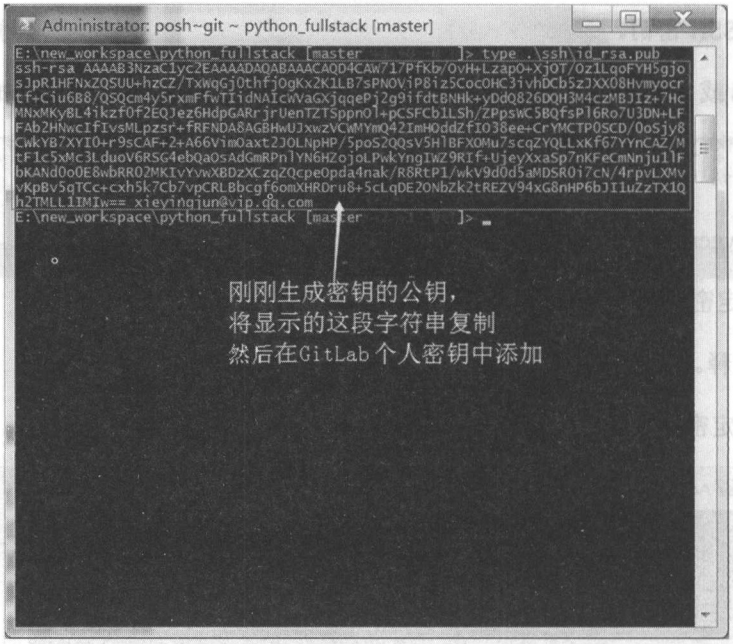


图 6-62

（12）在 GitLab 中添加 SSH 公钥。

登录 GitLab，单击右上角个人头像，选择 Settings 设置选项。在设置页面选择 SSH Keys 标签，在 Key 中粘贴第 10 步显示的公钥，保存这组密钥，如图 6-63 所示。

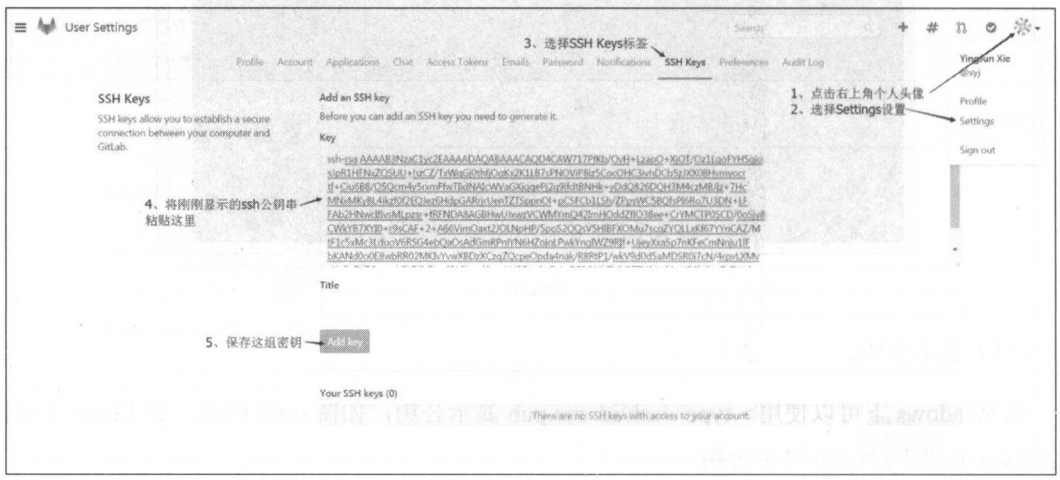


图 6-63

（13）在本地版本库中添加远程库地址。

前面我们修改了容器的 SSH 地址，把原来的 22 端口和宿主机 6050 端口绑定了，所以这里的配置方法与正常添加远程仓库的方法略有区别，如图 6-64 所示。

```
    > git remote add home git@192.168.1.137:xyj/python_fullstack.git
# 添加远程仓库
    > git remote set-url home
ssh://git@192.168.1.137:6050/xyj/python_fullstack.git  # 修改远程
    # 仓库的 URL，因为通信的 SSH 从 22 端口变成了 6050 端口
```

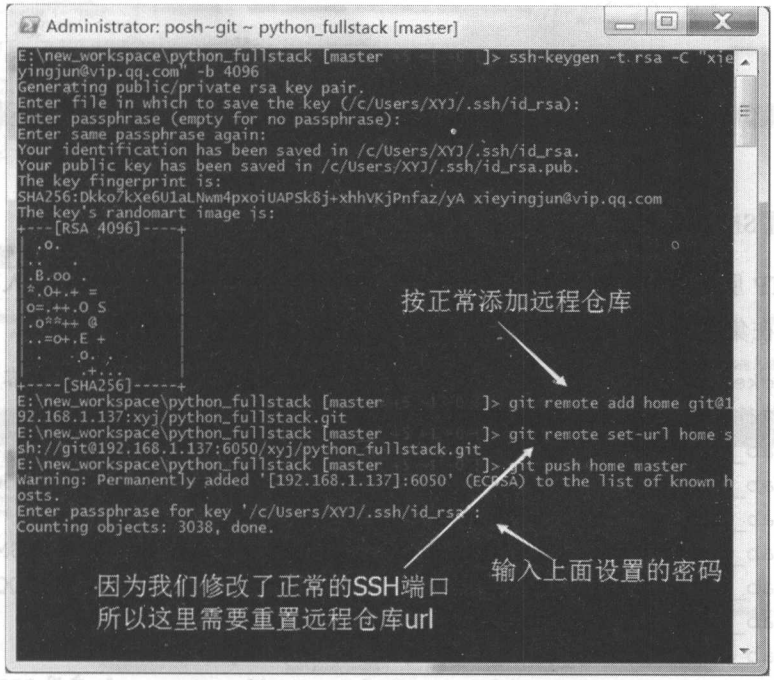


图 6-64

（14）获取 QQ 邮箱授权码。

登录 QQ 邮箱 https://mail.qq.com/，获取 QQ 邮箱授权码，如图 6-65 所示。其他邮箱大同小异。

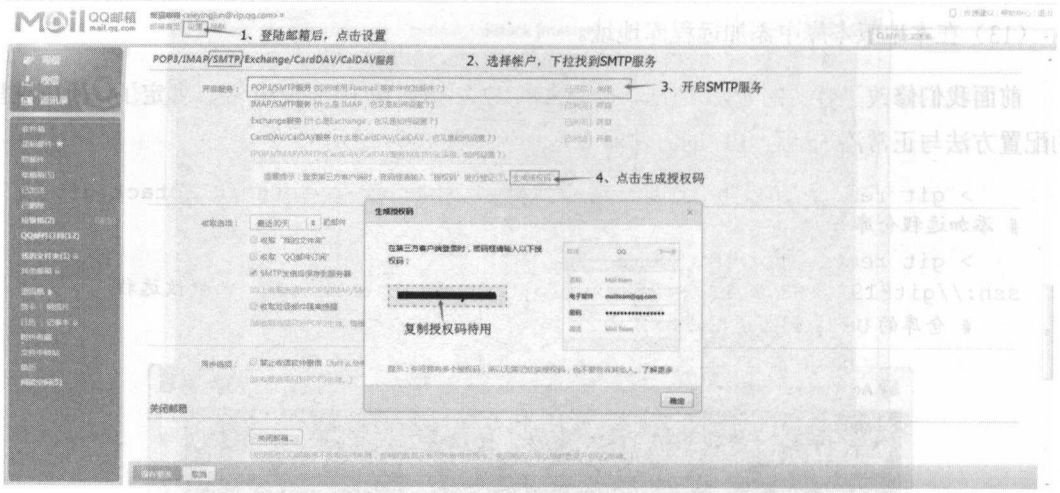


图 6-65

（15）设置 SMTP 服务。

设置 SMTP 服务时需要修改/etc/gitlab/gitlab.rb 文件，我们通过在终端中输入下列命令与容器进行交互，在集群模式下可以使用 yml 配置文件进行预设置，如图 6-66 所示。

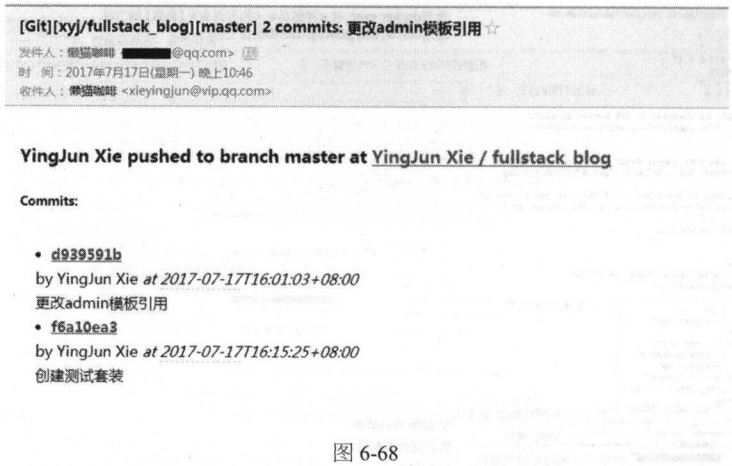
```
    $ docker exec -it c-gitlab vi /etc/gitlab/gitlab.rb  # 与容器 c-gitlab
# 交互，执行 vi 编辑/etc/gitlab/gitlab.rb 配置文件
    gitlab_rails['smtp_enable'] = true  # 使用 SMTP 发送邮件
    gitlab_rails['smtp_address'] = "smtp.qq.com"  # SMTP 服务器地址
    gitlab_rails['smtp_port'] = 465  # SMTP 端口
    gitlab_rails['smtp_user_name'] = "*******@qq.com"  # 登录 SMTP 邮箱账号
    gitlab_rails['smtp_password'] = "**********"  # 前面获取的 QQ 邮箱授权码
    gitlab_rails['smtp_domain'] = "smtp.qq.com"  # SMTP 域
    gitlab_rails['smtp_authentication'] = "login"  # 验证方式
    gitlab_rails['smtp_enable_starttls_auto'] = true  # 启用 STARTTLS
    gitlab_rails['smtp_tls'] = true  # 使用 TLS 访问 SMTP
    gitlab_rails['gitlab_email_from'] = '*******@qq.com' # 发件人地址
    $ docker exec -it c-gitlab gitlab-ctl reconfigure  # 重新加载 GitLab 配置
```

图 6-66

（16）设置发送邮件地址，如图 6-67 所示。

图 6-67

（17）邮箱收到测试邮件，如图 6-68 所示。


[Git][xyj/fullstack_blog][master] 2 commits: 更改admin模板引用 ☆

发件人：懒猫咖啡 [redacted]@qq.com>

时 间：2017年7月17日(星期一) 晚上10:46

收件人：懒猫咖啡 <xieyingjun@vip.qq.com>

YingJun Xie pushed to branch master at YingJun Xie / fullstack_blog

Commits:

- d939591b
by YingJun Xie *at 2017-07-17T16:01:03+08:00*
更改admin模板引用
- f6a10ea3
by YingJun Xie *at 2017-07-17T16:15:25+08:00*
创建测试套装

图 6-68

第7章
数据库介绍

7.1 数据库简介

数据库是操作系统上的一个文件或一套数据处理系统，用于持久保存数据，数据以一定的数据结构存放在内，可以对数据进行创建、读取、更新和删除操作。

数据库（DataBase，DB）是一个长期存储在计算机内的、有组织、有共享、统一管理的数据集合。它是一个按数据结构来存储和管理数据的计算机软件系统。数据库的概念实际包含两层意思：

（1）数据库是一个实体，它是能够合理保管数据的“仓库”，用户在该“仓库”中存放要管理的事务数据，“数据”和“库”两个概念结合成为数据库。

（2）数据库是数据管理的新方法和技术，它能更合适地组织数据、更方便地维护数据、更严密地控制数据和更有效地利用数据。[1]

目前数据库类型主要分为关系型数据库和非关系型数据库两种。

2017年4月全球数据库排行榜[2]如图7-1所示。

---

1 数据库解释引用
http://baike.baidu.com/link?url=fCICoUqSPtN21QwRAnkmdCDczJ_aKEgmZcFMIzGIXEYkrYIT2Nhfo2hIrDinMZC77bGIjqnw20VFNA61hAYnUYM0BU1v0y9KPL9iRCR3oy6AwYOVGRkEcRsczP4gvZY5WZMj2sIxyStb1179DT20A_#1_4

2 数据来源 https://db-engines.com/en/ranking

330 systems in ranking, June 2017

Rank Jun 2017	Rank May 2017	Rank Jun 2016	DBMS	Database Model	Score Jun 2017	Score May 2017	Score Jun 2016
1.	1.	1.	Oracle	Relational DBMS	1351.76	-2.55	-97.49
2.	2.	2.	MySQL	Relational DBMS	1345.31	+5.28	-24.83
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1198.97	-14.84	+33.16
4.	4.	↑ 5.	PostgreSQL	Relational DBMS	368.54	+2.63	+61.94
5.	5.	↓ 4.	MongoDB	Document store	335.00	+3.42	+20.38
6.	6.	6.	DB2	Relational DBMS	187.50	-1.34	-1.07
7.	7.	↑ 8.	Microsoft Access	Relational DBMS	126.55	-3.33	+0.32
8.	8.	↓ 7.	Cassandra	Wide column store	124.12	+1.01	-7.00
9.	9.	↑ 10.	Redis	Key-value store	118.89	+1.44	+14.39
10.	10.	↓ 9.	SQLite	Relational DBMS	116.71	+0.64	+9.92

图 7-1

7.2 关系型数据库

关系型数据库是将数据存储在一系列的表中，表与表之间的关系通过外键表示。外键是表中一行数据对另外一张表中一行数据的唯一引用。主键由表中一个或多个数据段组合而成，用于表中确定数据唯一性的约束，主键的值既不能重复，也不能为空。目前流行的关系型数据库主要有：Oracle、MySQL、Microsoft SQL Server、PostgreSQL、DB2、Microsoft Access、SQLite 等。

1. 数据库范式

数据库范式其实是指数据库设计规范化，通过规范化设计减少数据库中的数据冗余，提高数据的一致性。关系模型的发明者埃德加·科德（Edgar F. Codd）在 1970 年时定义了第一范式、第二范式和第三范式的概念，在 1974 年与 RaymondF.Boyce 共同定义了第三范式的改进范式——BC 范式。除此之外，数据库范式还包括针对多值依赖的第四范式、连接依赖的第五范式、DK 范式和第六范式。

现在，数据库设计最多满足 3NF（第三范式），普遍认为范式过高，虽然对数据关系有更好的约束性，但也导致数据关系表增加而令数据库 I/O 更易繁忙，原来交由数据库处理的关系约束现今更多是在数据库应用程序中完成。[1]

1 数据库范式解释：https://en.wikipedia.org/wiki/Database_normalization

（1）第一范式。

第一范式定义了每一个属性的值只能是单个值。简单来说，就是要求一列的值在一行中只能是一个，如图 7-2 所示。

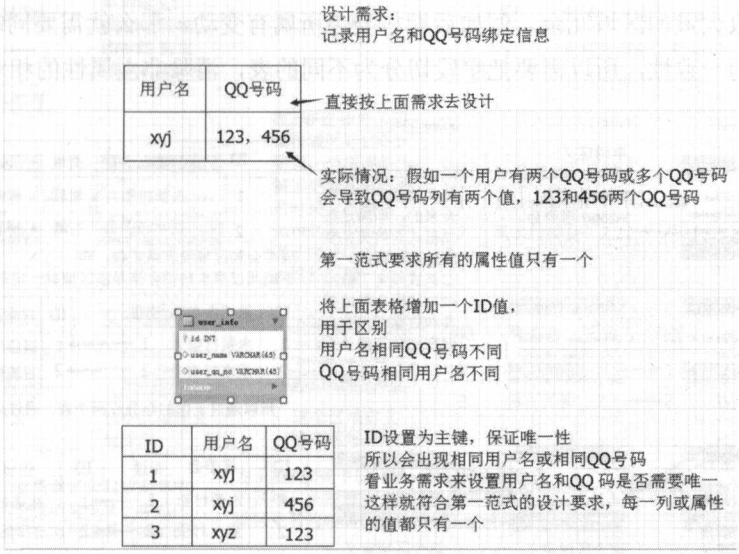


图 7-2

上面例子如果只是按照业务需求直接设计数据库，就会出现同一个用户名可能存在两个或多个 QQ 号码的情况，这不符合第一范式中要求的每项属性的值都要唯一，所以需要将两个 QQ 号码拆分成两行数据。这里又产生了一个新问题，同一个用户名不同 QQ 号码或者同一个 QQ 号码不同用户名如何区别开来呢？所以又需要增加一个新的字段 ID，通过将 ID 列设为主键来保证该值的唯一性，从而区分每一行数据的唯一性。至于用户名或 QQ 号码是否唯一，则要按实际业务需求来定。

（2）第二范式。

第二范式是在第一范式的基础上扩展开来的，即满足第二范式的前提是要先满足第一范式。第二范式规定了数据表内的所有数据都要和数据表内的主键有完全依赖关系。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分，通常需要

为表加上一列，以存储各个实例的唯一标识。简而言之，第二范式就是在第一范式的基础上属性完全依赖于主键。[1]

这里以一个笔者之前项目中用到的数据表设计为例来说明。有这样一个需求：每个村委会所属一个街道，每个街道属于一个区域，这样的内容关联起来，使用第二范式设计后如图 7-3 所示。每个村委会只属于一个街道，每个区域由多个街道组成。如果把所有字段都写在同一个表内，则会导致大量的区域冗余。假如后期村委会所属有变动，那么就需要同时修改多个字段才能保证数据的一致性，所以需要把字段切分为不同的表，消除非主属性的相对依赖。

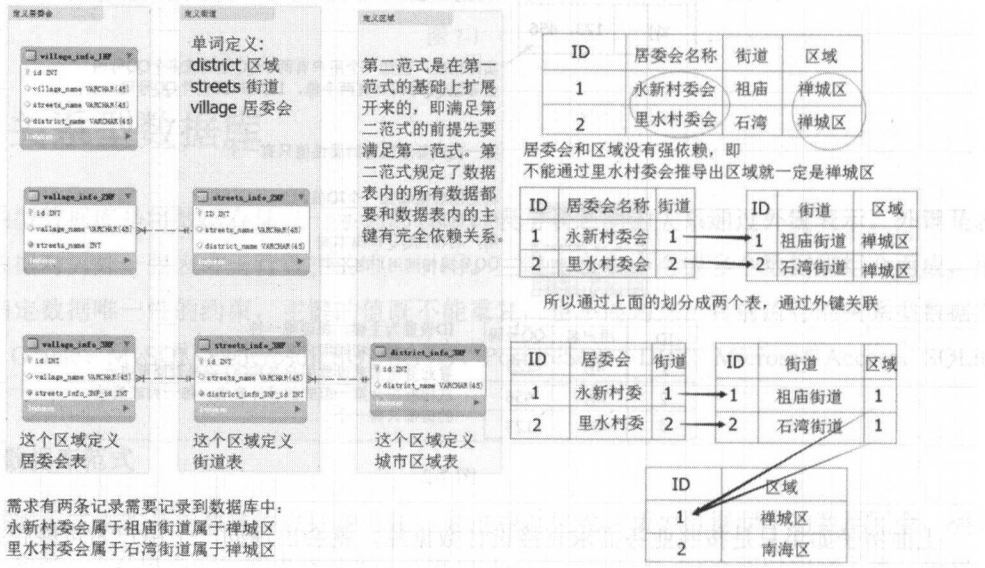


图 7-3

（3）第三范式。

第三范式要求一个数据库表中不包含在其他表中已包含的非主关键字信息。[2]

---

1 第二范式解释部分引用
http://baike.baidu.com/link?url=EnJNTnBfIkQemL_BX4FfVhJudXumt7Zsfw3gSctOHF6RdMr5xwWHF3isJ_uj0HxDrh6O5eNZwRp8iuqbJ1SqjnUSLT-vl4Q8MZB9Qc3pKnkyfIfc50KIg_78m3RqEO3i5q_rZRHUJL7jzbOBpcyb-a#2_2

2 第三范式解释引用
http://baike.baidu.com/link?url=Xh23B9J01LwyjhRsrDZxDAKpvengl29Iu3_2wG9_Bqeiz0mHU4Yqaf8q3427PS7d8FX11uGmcKIc-6OZnJbnyq

前面第二范式中，我们将表切分成了村委会和街道两个表，假如禅城区有 100 个街道，当需要给禅城区更名的时候，按第二范式划分的数据表在执行更新时需要更新 100 条记录。第三范式是在第二范式的基础上消除非主属性的传递依赖，即如图 7-4 所示，将街道和区域再划分成两个表，当需要给禅城区更名时，只需要更新 1 条记录即可。

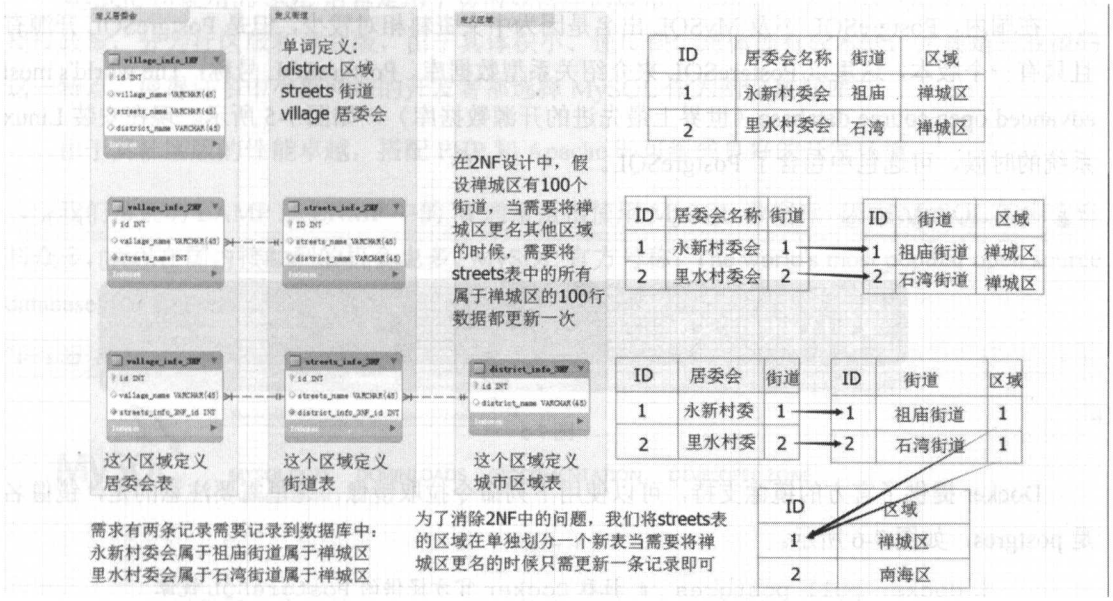


图 7-4

（4）第四范式、第五范式、第六范式。

一般情况下，数据库设计能满足第三范式设计即可解决商业应用问题，超过第三范式的设计会导致系统的 I/O 读取操作过于频繁，所以知道有第四五六范式的概念就足够了。若想了解这些范式的更多信息，则可以到网站 https://en.wikipedia.org/wiki/Database_normalization 查看相关内容定义。

PostgreSQL 数据库

PostgreSQL 是一个功能强大的开源对象关系数据库系统。拥有超过 15 年的积极发展和成熟的架构，在可靠性、数据完整性和正确性方面赢得了良好的声誉。它运行在所有主要操作系统上，包括 Linux、UNIX（AIX、BSD、HP-UX、SGI IRIX、MacOS、Solaris、Tru64）和 Windows。它完全符合 ACID 标准，完全支持外键、连接、视图、触发器和存储过程（多种语言）。它支持大多数符合 SQL：2008 标准的数据类型，包括 INTEGER（整数型）、NUMERIC（数字型）、

BOOLEAN（布尔值）、CHAR（字符型）、VARCHAR（变长字符型）、DATE（日期）、INTERVAL（时间）和 TIMESTAMP（时间戳）。它还支持存储 binary large objects（二进制对象），包括图片、声音或视频。它具有用于 C/C ++、Java、.Net、Perl、Python、Ruby、Tcl 及 ODBC 等本地编程接口。[1]

在国内，PostgreSQL 不及 MySQL 出名是因为中文资料相对较少，但是 PostgreSQL 开源有且只有一个版本，这里以 PostgreSQL 来介绍关系型数据库。PostgreSQL 号称：The world's most advanced open source database（世界上最先进的开源数据库），如图 7-5 所示。现在安装 Linux 系统的时候，可选包中包含了 PostgreSQL。

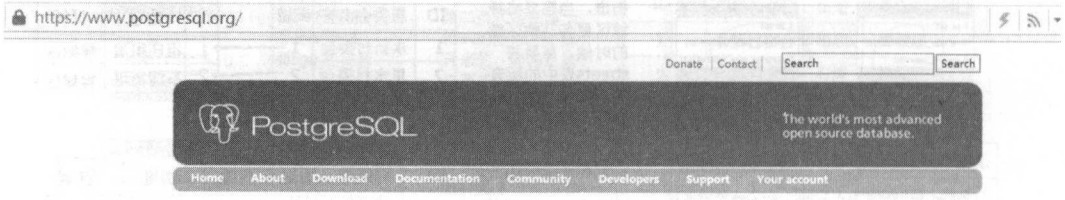


图 7-5

Docker 提供了官方的镜像支持，可以使用下列命令拉取镜像。这里需要注意的是，镜像名是 postgres，如图 7-6 所示。

```
$ docker pull postgres  # 拉取 Docker 官方提供的 PostgreSQL 镜像
```

```
xyj@xyj-home:~$ docker pull postgres
Using default tag: latest
latest: Pulling from library/postgres
9f0706ba7422: Pull complete
b7f935097c1d: Pull complete
9d3be652286b: Pull complete
93ca38c967a2: Pull complete
757f9036ad24: Pull complete
2278e49211f1: Pull complete
717785ad8220: Pull complete
63d5177d701b: Pull complete
b71fb03d2bdf: Pull complete
e8f2a0b376a0: Pull complete
c9686ffff9de: Pull complete
ba508d242abc: Pull complete
d8f09ded3eec: Pull complete
Digest: sha256:6d2f933d41ecf85d6291c5ae855caa59f1040c82c3adbfdca5d6ba60b8d37613
Status: Downloaded newer image for postgres:latest
```

docker官方提供PostgreSQL的镜像，这里需要主要的是镜像名是postgres

图 7-6

MySQL 数据库

MySQL 是一个关系型数据库管理系统，由瑞典 MySQL AB 公司开发，目前属于 Oracle 旗下产品。MySQL 是最流行的关系型数据库管理系统之一，在 Web 应用方面，MySQL 是最好的

1 PostgreSQL 解释源于官方简介 https://www.postgresql.org/about/

RDBMS（Relational Database Management System，关系数据库管理系统）应用软件。

MySQL 是一种关系型数据库管理系统，关系型数据库将数据保存在不同的表中，而不是将所有数据放在一个大仓库内，这样就提高了速度和灵活性。

MySQL 所使用的 SQL 语言是用于访问数据库的最常用标准化语言。MySQL 软件采用了双授权政策，分为社区版和商业版，由于其体积小、速度快、总体拥有成本低，尤其是开放源码这一特点，使得许多中小型网站的开发者都选择 MySQL 作为网站数据库。

由于其社区版的性能卓越，搭配 PHP 和 Apache 即可组成良好的开发环境。[1]

我们常说的 LAMP 或 LNMP 中的 M 通常是指使用 MySQL 数据库，因为 MySQL 的中文资料众多，所以在国内接触到的机会也多。MySQL 官方号称：The world's most popular open source database（世界上最流行的开源数据库），如图 7-7 所示。

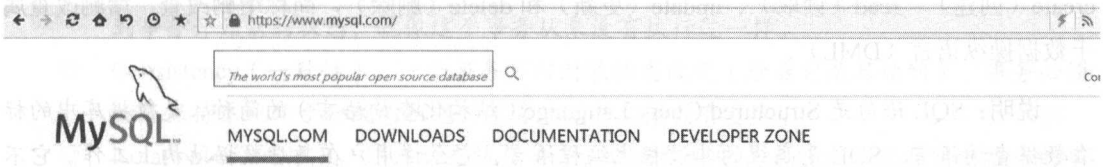


图 7-7

Docker 也提供了官方的镜像支持，可以使用下列命令拉取镜像，如图 7-8 所示。

```
$ docker pull mysql  # 拉取 docker 官方提供的 MySQL 的镜像
```

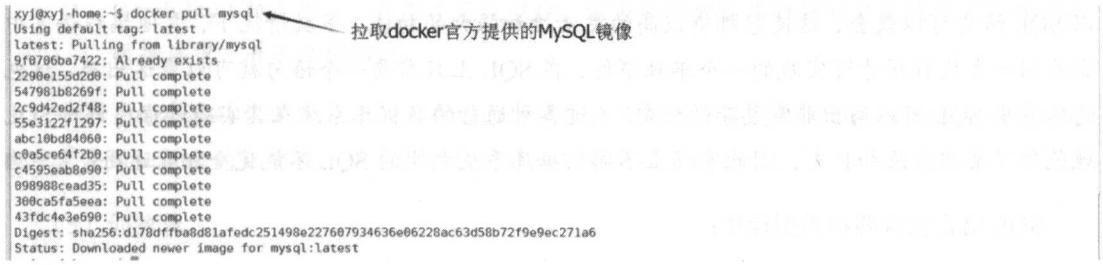


图 7-8

Oracle 数据库

Oracle 是全球最大的信息管理软件及服务供应商[1]，这里我们说的 Oracle 主要是指 Oracle

1　Mysql 解释源于 http://baike.baidu.com/item/mySQL/471251

Database，目前最新版本是 12c，这里的 c 是指 clouds，专门为云设计的版本。甲骨文公司为 Oracle 12c 设计了新的 Multitenant 架构（多租户架构），In-Memory 列存储以及对 JSON 文档的支持。

Oracle Database 是商业软件，是目前市场占用率第一的数据库软件，但是授权费昂贵，所以之前阿里巴巴才提出去 IOE 概念。IOE 是指 IBM 的小型机、Oracle 的数据库软件以及 EMC 存储设备。

SQL 基本语法

记得一个笑话，有个人应聘工程师，HR 问：你会什么？答：我玩数据库很溜。HR 问：怎么溜法呀？答：增删改查。

通常我们说增删改查主要是指使用 SQL 语句在数据库的一系列操作简称，增删改查来源于 create（创建）、read（读取）、update（更新）和 delete（删除），简称增删改查，增删改查属于数据操纵语言（DML）。

说明：SQL 语句是 Structured Query Language（结构化查询语言）的简称，是数据库中的标准数据查询语言。SQL 是高级的非过程化编程语言，它允许用户在高层数据结构上工作。它不要求用户指定对数据的存放方法，也不需要用户了解其具体的数据存放方式。而它的界面，能使具有底层结构完全不同的数据库系统和数据库之间，使用相同的 SQL 作为数据的输入与管理。它以记录项目（records）的合集（set）作为操纵对象，所有 SQL 语句接受项集作为输入，回提交的项集作为输出，这种项集特性允许一条 SQL 语句的输出作为另一条 SQL 语句的输入，所以 SQL 语句可以嵌套，这使它拥有极高的灵活性和强大的功能。多数情况下，其他编程语言中需要用一大段程序才可实践的一个单独事件，在 SQL 上只需要一个语句就可以表达出来。这也意味着用 SQL 可以写出非常复杂的语句。不过各种通行的数据库系统在其实践过程中都对 SQL 规范作了某些编改和扩充，因此实际上不同数据库系统之间的 SQL 不能完全相互通用。[2]

SQL 语言包含四种类型操作：

◎ Data Control Language（数据控制语言，DCL）：使用 grant（授权）和 revoke（撤销

1 Oracle 解释源于 http://baike.baidu.com/link?url=nEt5PotU1qOh0DbvmJjzNIv1Tjb7l9qKtbx0N-2gdMG_lJ9zApw8JJqAgq81FzqtinleWTR1qEeTDeP1mBrEjK

2 SQL 语言解释引用 https://en.wikipedia.org/wiki/SQL

授权）两个指令定义数据访问的权限。

◎ Data Manipulation Language（数据操纵语言，DML）：使用 insert（插入）、select（查询）、update（更新）和 deleted（删除），即“CRUD”四种指令来对数据进行操作。

◎ Data Definition Language（数据定义语言，DDL）：使用 create（创建）、alter（更改）和 drop（删除表）三种指令来定义数据结构和数据库对象。

◎ Transaction Control Language（事务控制语言，TCL）：主要使用 commit（提交）、rollback（回滚）和 savepoint（保存点）三种指令来控制事务状态。

说明：事务是指由一个用户执行一条或多条 SQL 语句后，数据从锁定到解锁完成的一系列操作，这一过程称为事务。使用事务控制可以保证数据的正确性。事务的基本要素可简称为 ACID，具体如下。

◎ Atomicity（原子性）。整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中如果发生错误，会被 rollback（回滚）到事务开始前的状态，就像这个事务从来没有执行过一样。

◎ Consistency（一致性）。一个事务可以封装状态改变（除非它是只读的）。事务必须始终保持系统处于一致的状态，不管此刻并发事务有多少。

◎ Isolation（隔离性）。隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行相同的功能，那么事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请求，使得在同一时间仅有一个请求用于同一数据。

◎ Durability（持久性）。在事务完成以后，该事务对数据库所作的更改便持久地保存在数据库中，并不会被回滚。

一个支持事务的数据库，必须具有这四种特性，否则在事务过程（Transaction processing）当中无法保证数据的正确性，交易过程极可能达不到交易方的要求。[1]

（5）创建表。

关系型数据库中表的概念类似于我们看到的 Excel 表格，由行列组成的一张表反映了数据间的关系。每一列都对应一个数据类型，通过不同的数据类型去约束该列值的范围类型。每一行通过主键或约束来标识唯一，通过唯一识别后的其他列中的值为该行属性。下面演示使用 SQL

1　部分解释引用 http://baike.baidu.com/item/acid/10738#viewPageContent

语句针对 PostgreSQL 数据库，其他数据库更改相应数据类型即可，如图 7-9 和图 7-10 所示。

```
CREATE TABLE district_info_3NF(  # 创建一个 district_info_3NF 的表
  id SERIAL PRIMARY KEY ,  # 设置列名为 ID，数据类型为连续增长型
  district_name TEXT  # 设置列名为 district_name，区域名称类似 TEXT
);  # 一定要用分号“;”养成每写完一段 SQL 语句都在结尾标上分号的习惯，分号之间为一段
# 执行指令
```

```
CREATE TABLE district_info_3NF(
  id SERIAL PRIMARY KEY ,
  district_name TEXT
);

CREATE TABLE streets_info_3NF(
  id SERIAL PRIMARY KEY ,
  streets_name TEXT,
  district_info_3NF_id INTEGER REFERENCES district_info_3NF(id)
);

CREATE TABLE vallage_info_3NF(
  id SERIAL PRIMARY KEY ,
  vallage_name TEXT,
  streets_info_3NF_id INTEGER REFERENCES streets_info_3NF(id)
);
```

自增长序列

postgresql中创建示范中的第三范式的三个数据表

使用district表的id做streets表的外键

图 7-9

```
CREATE SCHEMA IF NOT EXISTS `mydb` DEFAULT CHARACTER SET utf8 ;
USE `mydb` ;

-- Table `mydb`.`district_info_3NF`
CREATE TABLE IF NOT EXISTS `mydb`.`district_info_3NF` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `district_name` VARCHAR(45) NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB;

-- Table `mydb`.`streets_info_3NF`
CREATE TABLE IF NOT EXISTS `mydb`.`streets_info_3NF` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `streets_name` VARCHAR(45) NULL,
  `district_info_3NF_id` INT NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_streets_info_3NF_district_info_3NF1_idx` (`district_info_3NF_id` ASC),
  CONSTRAINT `fk_streets_info_3NF_district_info_3NF1`
    FOREIGN KEY (`district_info_3NF_id`)
    REFERENCES `mydb`.`district_info_3NF` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- Table `mydb`.`vallage_info_3NF`
CREATE TABLE IF NOT EXISTS `mydb`.`vallage_info_3NF` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `vallage_name` VARCHAR(45) NULL,
  `streets_info_3NF_id` INT NOT NULL,
  PRIMARY KEY (`id`),
  INDEX `fk_vallage_info_3NF_streets_info_3NF1_idx` (`streets_info_3NF_id` ASC),
  CONSTRAINT `fk_vallage_info_3NF_streets_info_3NF1`
    FOREIGN KEY (`streets_info_3NF_id`)
    REFERENCES `mydb`.`streets_info_3NF` (`id`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

创建一个用户专属集合，字符集使用utf8

MySQL中创建第三范式中的表SQL语句

创建表district，注意要指定表所在的用户集合

MySQL中自增长的使用int类型定义一个auto_increment属性

主键为id列

使用引擎为InnoDB，默认引擎，可跟据需求更改

图 7-10

上面两个语句都是创建相同功能的表，但是使用不同的数据库，语法相应的地方需要更改。或者用 Python 编程中的 ORM 框架，将数据库中的表映射成一个类，通过操作类实现数据库的增删改查，目前比较流行的是使用 SQLAlchemy 库来实现。

（6）修改表。

在不删除表和表中数据的前提下可以使用 alter 语句来修改表结构，实现增加字段、修改字段类型、重命名字段和删除字段，如图 7-11 所示。

```
--修改居委信息表，增加一个联系电话字段--
ALTER TABLE vallage_info_3nf
   ADD vallage_phone TEXT NULL;
--修改居委信息表，将联系电话字段类型由 TEXT 修改为 INTEGER--
ALTER TABLE vallage_info_3NF
   ALTER COLUMN vallage_phone TYPE INTEGER;
--修改居委信息表，将联系电话字段名称改为 phone--
ALTER TABLE vallage_info_3NF
   RENAME COLUMN vallage_phone TO phone;
--修改居委信息表，删除联系电话字段--
ALTER TABLE vallage_info_3NF
      DROP phone;
```

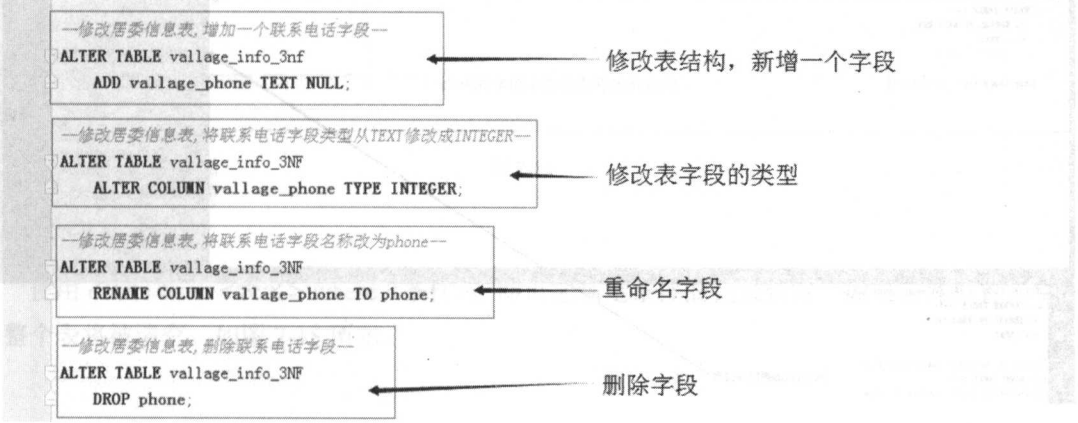


图 7-11

（7）删除表。

先创建一个删除示例的 test 表，再用 drop table <表名>删除指定表，如图 7-12 和图 7-13 所示。

```
CREATE TABLE test(
id SERIAL PRIMARY KEY ,
test TEXT
);  --创建 test 临时表
DROP TABLE test;  --删除 test 表
```

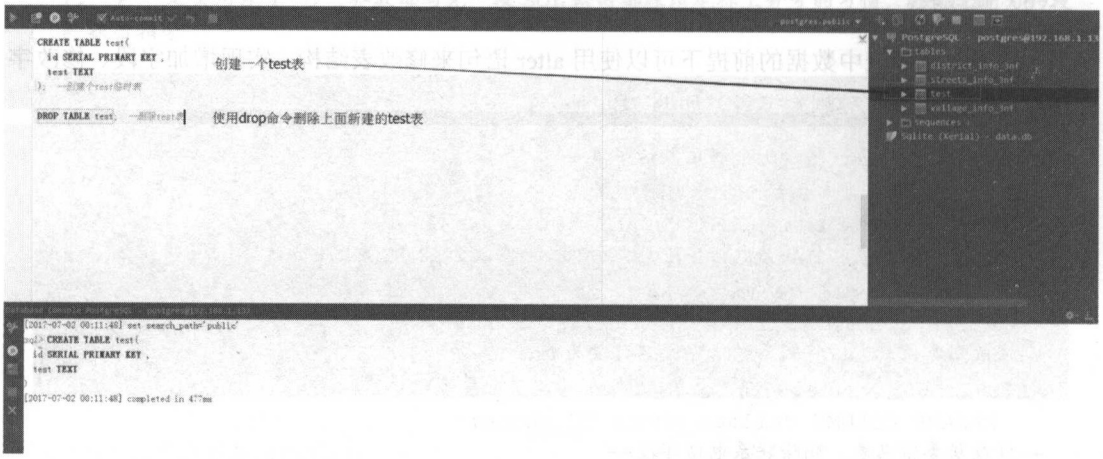


图 7-12

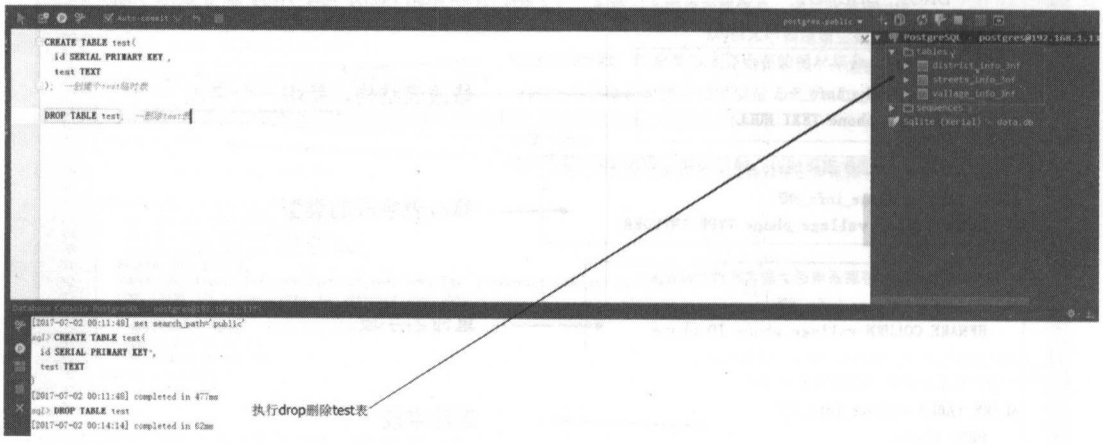


图 7-13

（8）新增语句。

前面讲的“增”是指向数据库的表中插入新数据，使用 insert into <table(表名)>(col1(列名 1),col2(列名 2),...) values(‘values1(往列 1 插入的值)’,’values2(往列 2 插入的值)’,...) ……，语法如下：

```
INSERT INTO district_info_3NF(district_name)VALUES ('禅城区');
```

（9）查询语句。

使用 select <显示字段,使用*号则显示全部字段> from <表名> [where <条件>]，查询这个表中的记录，可以在表名后使用 where 来增加过滤条件。这里查询街道表的所有记录，显示所有字段，如图 7-14 所示。

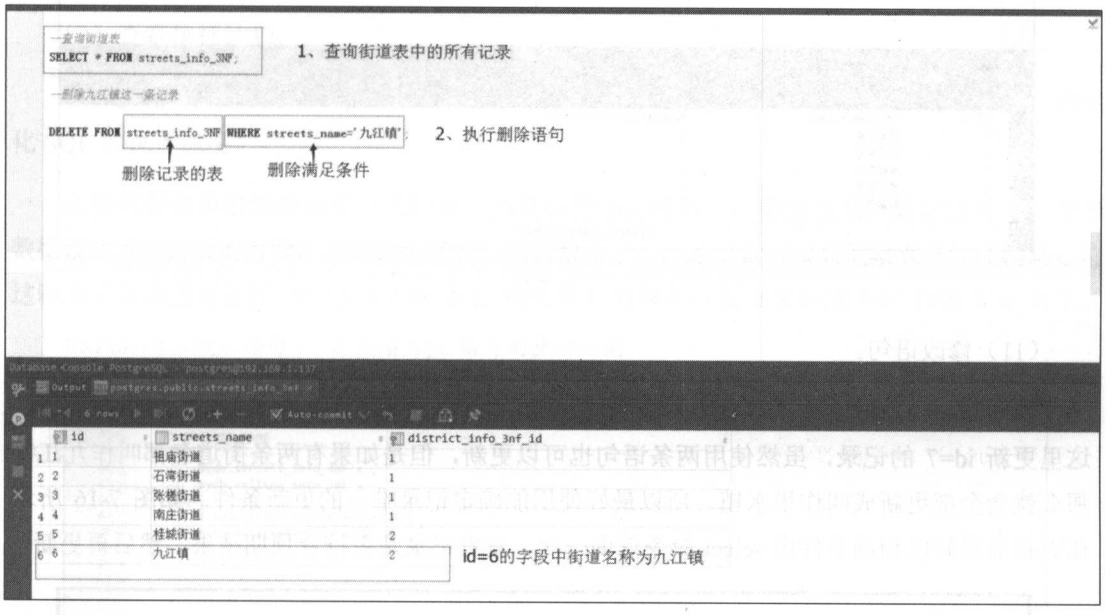


图 7-14

（10）删除语句。

使用 delete from <表名> where <条件>，即可把满足条件的记录删除，如果条件位置为空，则整个表将被清空。如图 7-15 所示。

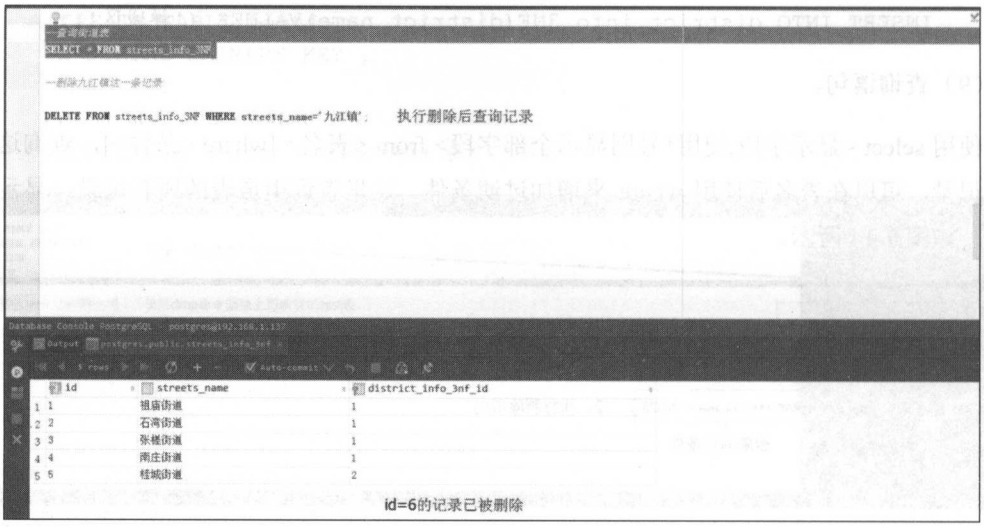


图 7-15

（11）修改语句。

使用 update <表名> set <列名>=<新值>[, <列名>=<新值>] [where <条件>]，即可更新记录。这里更新 id=7 的记录，虽然使用两条语句也可以更新，但是如果有两条街道名都叫作九江镇，那么就会全部更新成叫作里水镇，所以最好使用能确定记录唯一的更新条件。如图 7-16 所示，在更新前最好以相同条件用 select 命令查询一次，看看记录是否符合预期结果，然后再更新。

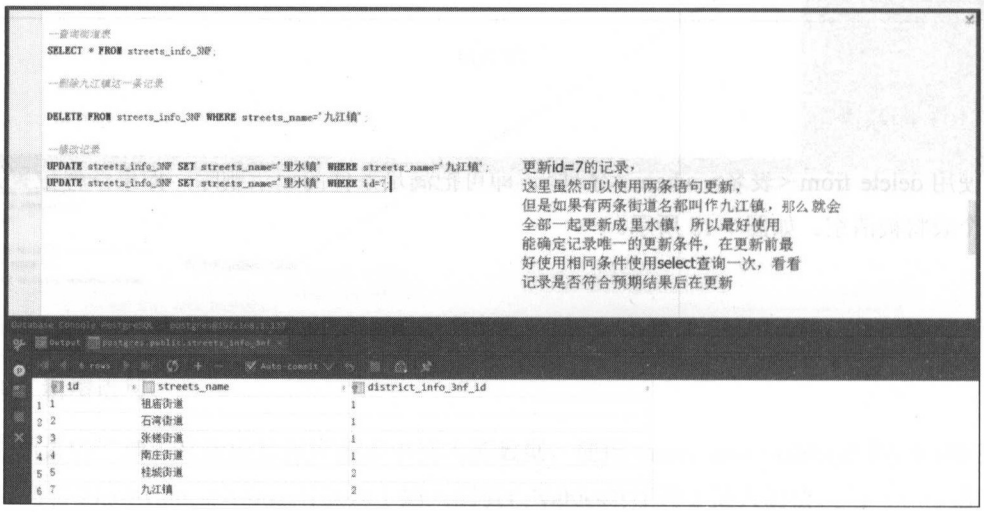


图 7-16

2. SQL 优化

（1）优美书写。

很多时候我们编写的 SQL 语句并不是只给自己看的，即便是给自己看的一段时间后也需要修改，也会遇到需要理清思路的情形，所以遵循一定方式去编写 SQL 语句有利于后期的阅读和编辑。每一行只放一个表、列、条件，这样阅读起来条理清晰。

（2）查看执行计划。

在 SQL 执行语句前加 explain 可以查看当前 SQL 语句执行的计划，这个计划可以让我们优化 SQL 的执行速度。

在所有查询中最慢的就是全表扫描，当数据量小的时候，还能勉强毫秒级出结果，但是当数据量达到千万级的时候，这时候跑全表扫描就很慢了，而通走索引或其他限制条件能精确定位这种方式查询速度最快。用 EXPLAIN 查看 SQL 语句的执行计划分别如图 7-17 和图 7-18 所示。

```
EXPLAIN <SQL 语句>  # 查看 SQL 语句的执行计划
```

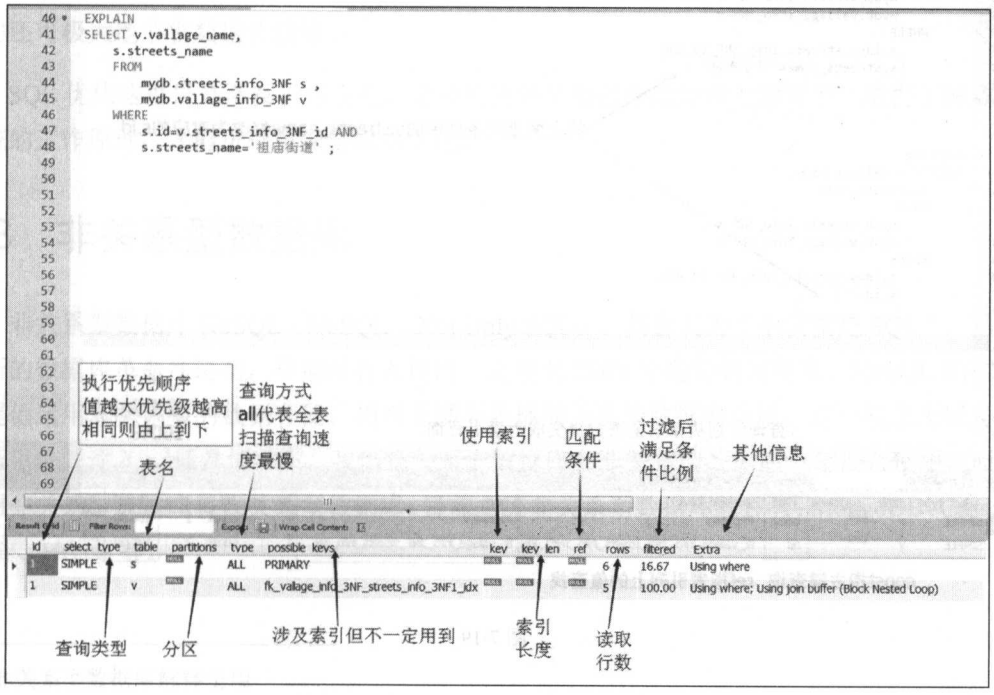


图 7-17

图 7-18

（3）调整查询条件。

通过简单调整查询条件，使查询从全表扫描转变成走索引查找。当数据量特别大的时候，这种调整效果非常明显，如图 7-19 所示。

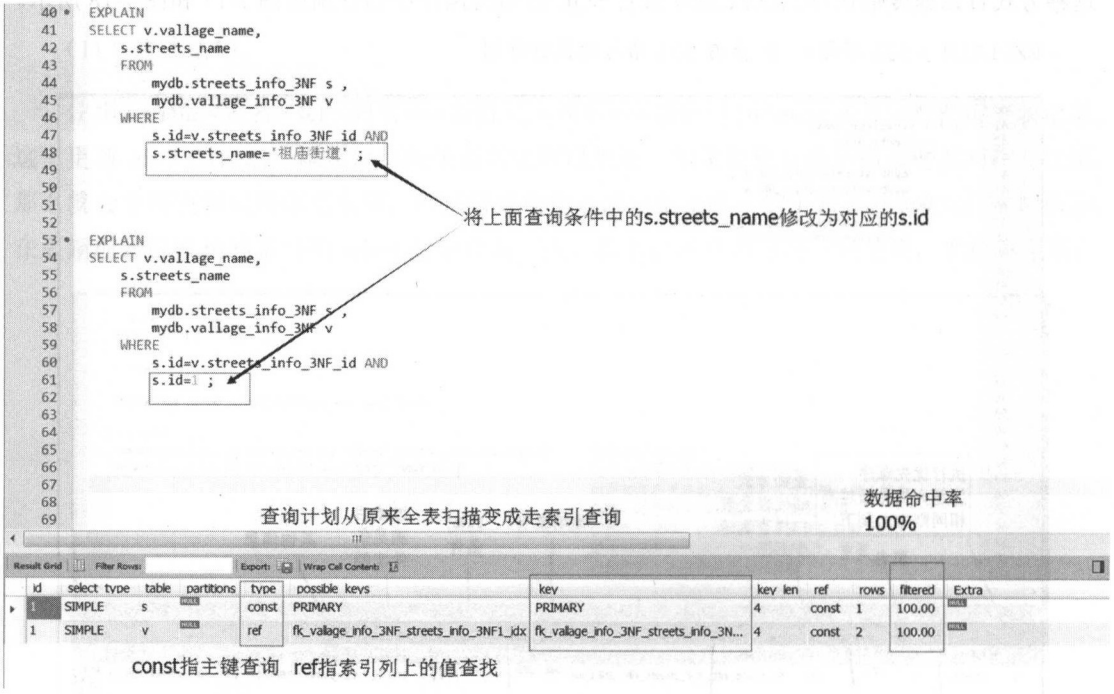


图 7-19

（4）创建索引。

很多时候我们会不假思索地在需要查询的条件上创建索引，但是这样真能起到很好的作用吗？一般来说，创建索引是希望通过走索引过滤 80%以上的无用数据，快速得到小范围查询值。布尔型的字段不宜建立索引，因为每次能过滤掉的数据很少。而唯一性的字段建立哈希（Hash）索引比默认建立的 B 树索引查询更快。

（5）不要在查询条件上计算。

很多时候我们查询一个范围的数据，例如查询 2016 年的销售情况，SQL 语句可能会写成 select * from TRANS where year(TRANS_DATE) = '2016'

TRANS_DATE 这列做了索引，但是查询还是会全表扫描。正确查询语句应该类似这样写法：

```
    select id, ic_card_id, trans_type from TRANS where TRANS_DATE >=
to_date('2016-01-01','YYYY-MM-DD') and TRANS_DATE <
to_date('2017-01-01','YYYY-MM-DD')。
```

即查询不要直接用 select *，指明需要的字段这样能大大节省内存的使用量，如果是远程传输的还可极大地节省数据传输量。

SQL 优化这里只是涉及了点皮毛，更多的需要从数据库的原理上面着手，透彻了解数据库系统的工作原理，才能进行更深层级的优化。

7.3　非关系型数据库

非关系型数据库 NoSQL（NoSQL = Not Only SQL），其含义为“不仅仅是 SQL”，是一项全新的数据库革命性运动，早期就有人提出，发展至 2009 年趋势越发高涨。NoSQL 的拥护者们提倡运用非关系型的数据存储，相对于铺天盖地的关系型数据库运用，这一概念无疑是一种全新思维的注入。[1]通常情况下，我们是关系型数据库和非关系型数据库一起混合使用，而不是使用非关系型数据库替代关系型数据库。目前主流非关系型数据库软件有：MongoDB、Cassandra、Redis、HBase 等。

1　非关系型数据库解释引用
http://baike.baidu.com/link?url=5MgAZFVhiFBgR2ujwrsffpFdUR2M8fktzKkGkunOBmdiaP0njzqW-ANRp7b5rzinnrQZMLlZbgodPKrHru7zpK#1

非关系型数据库目前分成四大类型。

◎ 键值对（Key-Value）存储数据库：Redis
◎ 文档类型数据库：MongoDB
◎ 列存储数据库：HBase
◎ 图形数据库：GraphDB

1. MongoDB 数据库

MongoDB（来自于英文单词“Humongous”，中文含义为“庞大”）是一个跨平台的面向文档的数据库。MongoDB 被分类为 NoSQL 数据库，它避开了传统的基于表的关系数据库结构，支持具有动态模式的 JSON 类文档（MongoDB 调用格式 BSON），使得数据在某些类型的应用程序中的集成更加便捷。MongoDB 是根据 GNU Affero 通用公共许可证和 Apache 许可证组合发布的，是免费的开源软件。

MongoDB 起源于 2007 年纽约的一个创业项目 10gen，基于平台级服务 PaaS 构建一个用于开发、托管并具有自动缩放 Web 应用程序的在线服务，这个项目没做起来，但里面用的数据库反而流行起来，后来就改名为 MongoDB。此后，MongoDB 被许多主要网站和服务采用为后端软件，其中包括 Craigslist、eBay、Foursquare、SourceForge、Viacom 和纽约时报等。MongoDB 是最受欢迎的 NoSQL 数据库系统。[1]

MongoDB 的特点如下。

◎ 数据基于文档数据模型，而文档又基于 JSON 格式。后来 MongoDB 又开发了 BSON 格式，但是两者没什么区别。
◎ 在关系型数据库中，存储对象有表的概念，但在 MongoDB 中没有表的概念，对应的是集合 collection 的概念。
◎ 即席查询用户可根据自己的需求，灵活地选择查询条件，系统能够根据用户的选择生成相应的统计报表。即席查询与普通应用查询最大的不同是，普通的应用查询是定制开发的，而即席查询是由用户自定义查询条件的。[2]
◎ 支持索引和支持主从复制。

---

1 MongoDB 解释引用 wikipedia.org/wiki/MongoDB

2 即席查询概念引用
http://baike.baidu.com/link?url=SgQCErD1DG-Vy9RVyl4UStshW1Bl8jdxnvuf6t-BjlgyvkuXXWHtCD53oA7zmxnzD6U26OM_c2MLy5cDF8aVfmBY1y2ewTAgSuSVWvXr3quk3Zxqt1NcYDOAV-oDcKkl

◎　高伸缩性，在业务发展的不同时期，不管横向扩展还是纵向扩展，提高服务器配置或分布式部署对于 MongoDB 都是很容易实现的事情。

（1）拉取 MongoDB 镜像。

使用 Docker pull 命令拉取由 Docker 官方提供的 MongoDB 镜像，这里需要注意的是，拉取的名字是 mongo 而不是 MongoDB，如图 7-20 所示。

```
$ docker pull mongo  # 拉取 Docker 官方提供的 MongoDB 镜像
```

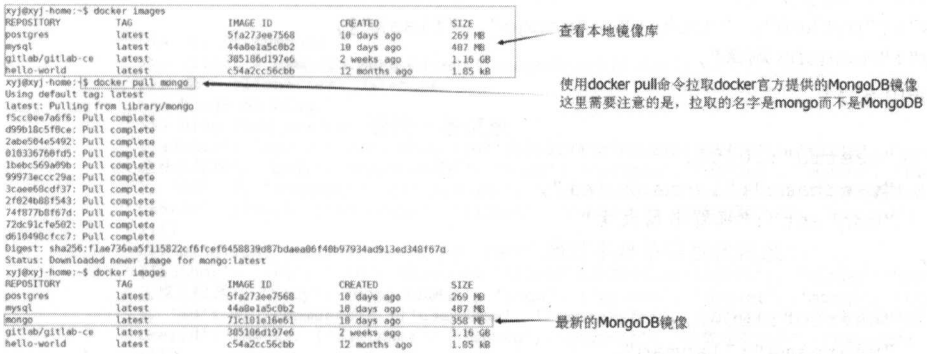


图 7-20

（2）启动 MongoDB。

拉取完镜像后使用下列代码即可启动 MongoDB 容器，如图 7-21 所示。

```
$ docker run -d \  # 参数-d 后台运行
> --name c-mongo \  # 设置容器别名
> -v /home/xyj/mongo_volume:/data/db \  # 将宿主机目录映射到容器的 MongoDB
# 数据保存位置
> -p 27017:27017 \  # 绑定宿主机 27017 端口
> mongo  # 选择启动镜像名称
```

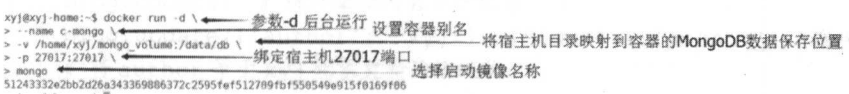


图 7-21

（3）pymongo 包的使用。

使用 MongoDB 官方提供的 pymongo 包构建连接 MongoDB，插入数据，如图 7-22 所示。

```
import pymongo # 引入 pymongo 包
client = pymongo.MongoClient('mongodb://192.168.1.137:27017/')
# 构造客户端
db = client.test_database  # 选择 test_database 数据库，假如没有这个数据库则自动
# 创建或使用 db = client['test_database']方式选择数据库
blog= db.blog  # 获取 blog 集合，没有集合则自动创建
post = {
    "id":3,
    "title":"python 全栈工程师",
    "author":"xyj",
    "tags":["python", "docker", "mongo", "flask"],
    "body":"MongoDB 测试",
    "comments":[
        {
            "userid":10001,
            "username":"lanmaokafei",
            "content":"祝贺书籍大卖"
        },
        {
            "userid":10002,
            "username":"lanmao",
            "content":"祝贺书籍大卖"
        }]
} # 构建一个新日志信息
blog.insert_one(post)
# 将日志 post 插入到 blog 集合中，insert_one 是插入一条记录
```

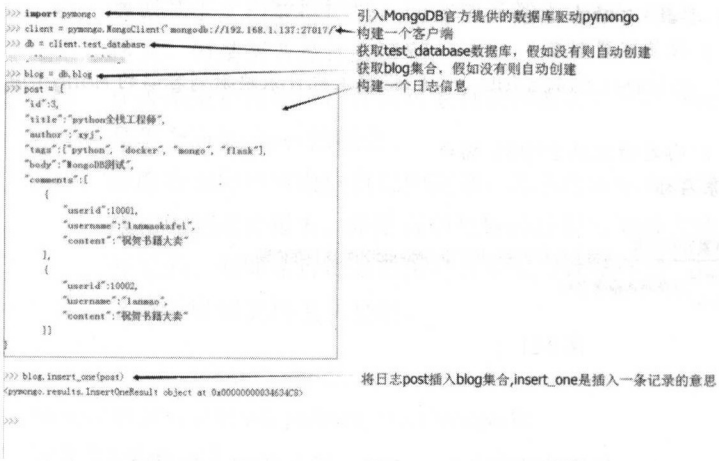


图 7-22

查找数据，如图 7-23 所示。

```
import pymongo # 引入 pymongo 包

client = pymongo.MongoClient('mongodb://192.168.1.137:27017/')
# 构造客户端
db = client.test_database  # 选择 test_database 数据库
blog= db.blog  # 选择 blog 集合
blog.find_one()  # 查找一条记录
blog.find_one({'author': 'xyj'})  # 通过条件查找一条记录
```

```
>>> import pymongo
>>> client = pymongo.MongoClient('mongodb://192.168.1.137:27017/')
>>> db = client.test_database
>>> blog= db.blog
>>> blog.find_one()← 查找一条记录
{'author': 'xyj', '_id': ObjectId('595a5f743c78d02aa4336776'), 'title': 'python
全栈工程师', 'body': 'MongoDB测试', 'tags': ['python', 'docker', 'mongo', 'flask'
], 'id': 3, 'comments': [{'username': 'lanmaokafei', 'content': '祝贺书籍大卖', '
userid': 10001}, {'username': 'lanmao', 'content': '祝贺书籍大卖', 'userid': 1000
2}]}
>>> blog.find_one({'author':'xyj'})← 通过字典条件查找记录
{'author': 'xyj', '_id': ObjectId('595a5f743c78d02aa4336776'), 'title': 'python
全栈工程师', 'body': 'MongoDB测试', 'tags': ['python', 'docker', 'mongo', 'flask'
], 'id': 3, 'comments': [{'username': 'lanmaokafei', 'content': '祝贺书籍大卖', '
userid': 10001}, {'username': 'lanmao', 'content': '祝贺书籍大卖', 'userid': 1000
2}]}
>>>
```

图 7-23

更新数据如图 7-24 所示。

```
import pymongo

client = pymongo.MongoClient('mongodb://192.168.1.137:27017/')
# 构造客户端
db = client.test_database # 选择 test_database 数据库
blog = db.blog  # 选择 blog 集合
blog.count()  # 统计集合内有多少记录
blog.update({'author': 'xyj'}, {"$set": {'author': '谢瑛俊'}})
# 将 author 等于 xyj 记录更新为谢瑛俊
   blog.find_one()
```

```
>>> import pymongo
>>> client = pymongo.MongoClient('mongodb://192.168.1.137:27017/')
>>> db = client.test_database
>>> blog= db.blog
>>> blog.count()
1
>>> blog.update({'author':'xyj'},{"$set":{'author':'谢瑛俊'}})
{'updatedExisting': True, 'ok': 1.0, 'nModified': 1, 'n': 1}
>>> blog.find_one()
{'comments': [{'content': '祝贺书籍大卖', 'userid': 10001, 'username': 'lanmaokaf
ei'}, {'content': '祝贺书籍大卖', 'userid': 10002, 'username': 'lanmao'}], '_id':
ObjectId('595a5f743c78d02aa4336776'), 'id': 3, 'title': 'python全栈工程师', 'auth
or': '谢瑛俊', 'body': 'MongoDB测试', 'tags': ['python', 'docker', 'mongo', 'flas
k']}
>>>
```

图 7-24

第 8 章

基于Flask开发Web项目

8.1 为项目创建虚拟环境

在开发中我们可能会用到不同版本的 Python，并且每个项目使用的包又不尽相同，所以我们使用 Python 的虚拟环境，这样可以使每个项目使用的 Python 相互独立，避免直接使用主环境 Python 而产生大量的包，不便于项目的移植迁移。

1. 手工创建虚拟环境

在 Python 3.3 之后官方把 virtualenv 包直接集成到内置包中，所以在 Python 3.3 之后创建虚拟环境，既可以用命令提示符，也可以在终端上使用下面命令手工创建虚拟环境。

```
    $python3 -m venv env  #第一个 venv 告诉我们需要使用 venv 这个包，第二个 env 是
# 创建
    #虚拟环境的目录，可以任意，很多时候直接就写成 venv，这里为了区分两个 venv，因此
# 把第二个写成了 env
    $ cd env/bin/  # 进入虚拟环境文件夹的 bin 目录
    $ source activate  # 激活 Python 虚拟环境
    $ deactivate  # 退出虚拟环境
```

在路径前显示（env）则代表现在处于 env 产生的虚拟环境中。在这里使用 pip 安装任何包都是与主环境 Python 隔离的，互不影响，如图 8-1 所示。

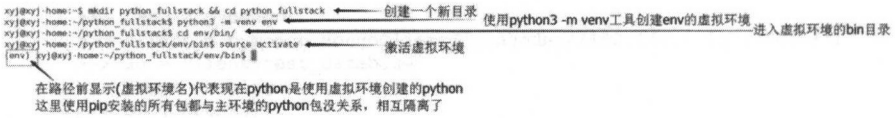


图 8-1

2．在 PyCharm 中创建虚拟环境

PyCharm 可以在项目创建的时候一并创建虚拟环境，具体参考第 3.4 节中的创建过程。当项目创建好后想修改当前使用的 Python 解释器，可以在设置中修改 Python 解释器。

按组合键 Ctrl+Alt+S 进入设置，单击 Project:<项目名>，选择 Project Interpreter（Python 解释器），单击右边齿轮，在显示的菜单中选择 Create VirtualEnv（创建新的虚拟环境），如图 8-2 所示。

图 8-2

8.2　快速搭建 HTTPS 网站应用

这里快速搭建 HTTPS 服务的项管技术并没采用 Nginx 或 Apache，而是直接用 Python 代码实现的 SSL 加密通信。

1. 生成密钥

使用 openssl 生成密钥和证书。

```
    $mkdir SSL_test  # 创建保存密钥的目录
    $cd SSL_test  # 进入 SSL_test 目录
    $openssl genrsa 1024 > ssl.key  # 生成密钥
    $openssl req -new -x509 -nodes -sha1 -days 365 -key ssl.key > ssl.cert
# 生成证书，有效期为 365 天
```

2. 测试部署

在该目录下新建一个 run.py 脚本，写入下列代码。

```
    #!/usr/bin/env python3
# encoding: utf-8

"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/11
"""
from werkzeug.serving import run_simple
from flask import Flask, render_template
import ssl

app = Flask(__name__)


@app.route('/')
def index():
    return render_template('index.html')


if __name__ == '__main__':
    ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)  # 选择加密版本
```

```
    ctx.load_cert_chain('ssl.cert', 'ssl.key')  # 加载证书和密钥
    run_simple('localhost', 4000, app, ssl_context=ctx)  # 通过 ssl_context
# 加载 ssl 对象
```

8.3 使用 PyCharm 在本机容器中开发

从 PyCharm 2016.1 Professional Edition 以上版本新增了开发工具，支持 Docker 和 Vagrant 部署开发。

在项目新建的时候指定远程Python解释器或者修改当前项目的解释器都能切换到容器中运行。这里演示使用 PyCharm 2017.1 Professional Edition 版将现有项目使用的 Python 解释器切换成容器中的 Python 解释器，使用 Docker 容器作为项目的运行测试环境。

（1）项目内容介绍。

首先创建 Dockerfile 和 Docker-compose.yml 两个文件，Dockerfile 用来构建用于程序调试的容器环境配置，Docker-compose.yml 用于运行由 Dockerfile 生成的容器配置。

```
Dockerfile File
# 指定引用官方提供 Python 3.6 版本镜像
FROM python:3.6

# 绑定宿主机 5000 端口
EXPOSE 5000

# 创建 app 目录
RUN mkdir /app

# 指定工作目录为 app
WORKDIR /app

# 复制本地 requirements.txt 到容器/app 下
COPY requirements.txt /app/requirements.txt

# 容器内运行命令，安装依赖包
RUN pip install -r requirements.txt

# 复制所有内容到/app 目录下
COPY . /app
```

```
# 使用 Python 解释器运行 flask-demo
CMD ["python", "flask-demo.py"]
```

如图 8-3 所示。

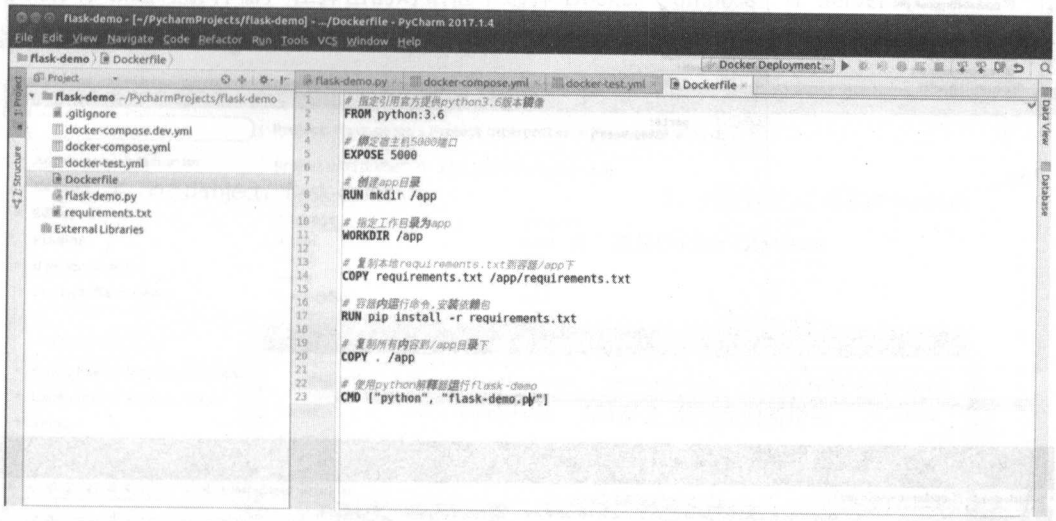


图 8-3

```
docker-compose.yml file
# Compose 文件格式版本
version: '2'

# 定义服务
services:

# Web 服务
  web:
    # 构建
    build: .
    # 宿主机与容器的端口映射
    ports:
     - "5000:5000"
```

如图 8-4 所示。

（2）在 Pycharm 中配置 Docker API 控制。

单击 File→Settings 或按组合键 Ctrl+Alt+S 调出设置面板，单击 Build, Execution, Deployme 选项，选择 Docker 设置，如图 8-5 所示。

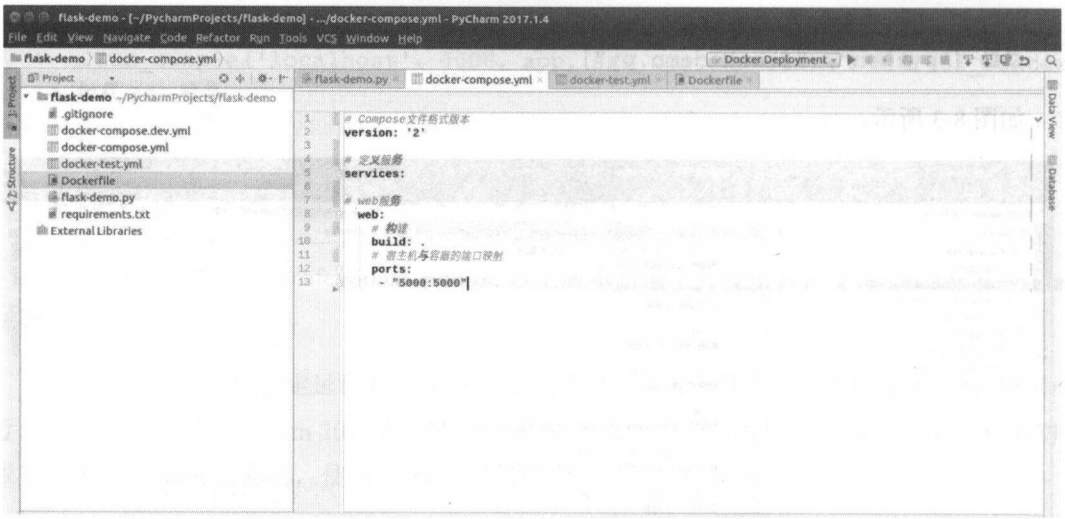

flask-demo - [~/PycharmProjects/flask-demo] - .../docker-compose.yml - PyCharm 2017.1.4
File Edit View Navigate Code Refactor Run Tools VCS Window Help
flask-demo > docker-compose.yml
Docker Deployment
Project
flask-demo ~/PycharmProjects/flask-demo
.gitignore
docker-compose.dev.yml
docker-compose.yml
docker-test.yml
Dockerfile
flask-demo.py
requirements.txt
External Libraries
flask-demo.py
docker-compose.yml
docker-test.yml
Dockerfile
Compose文件格式版本
version: '2'
定义服务
services:
web服务
web:
构建
build: .
宿主机与容器的端口映射
ports:
- "5000:5000"
Data View
Database
Structure
Project

图 8-4

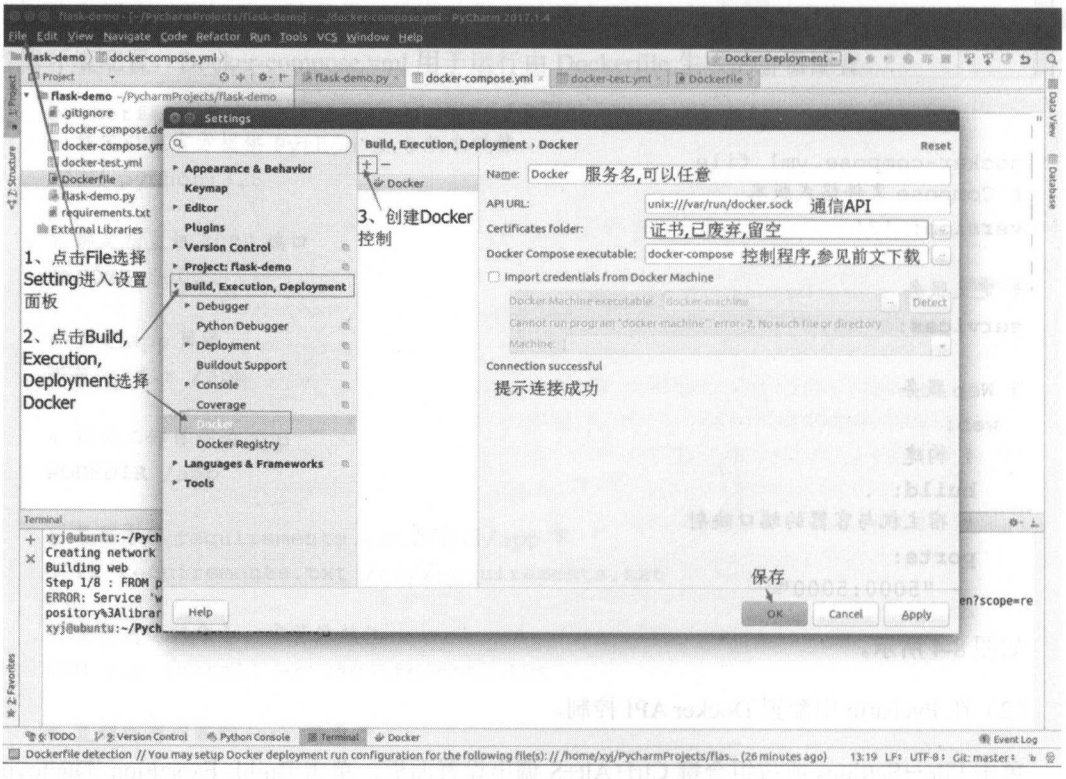

Settings
Build, Execution, Deployment › Docker
Reset
Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Project: flask-demo
Build, Execution, Deployment
Debugger
Python Debugger
Deployment
Buildout Support
Console
Coverage
Docker
Docker Registry
Languages & Frameworks
Tools
Docker
Name: Docker 服务名,可以任意
API URL: unix:///var/run/docker.sock 通信API
Certificates folder: 证书,已废弃,留空
Docker Compose executable: docker-compose 控制程序,参见前文下载
Import credentials from Docker Machine
Docker Machine executable: docker-machine
Detect
Cannot run program "docker-machine": error=2, No such file or directory
Machine:
Connection successful
提示连接成功
1、点击File选择Setting进入设置面板
2、点击Build, Execution, Deployment选择Docker
3、创建Docker控制
保存
Help
OK
Cancel
Apply
Terminal
Creating network
Building web
Step 1/8 : FROM p
TODO
Version Control
Python Console
Terminal
Docker
Event Log
Dockerfile detection // You may setup Docker deployment run configuration for the following file(s): // /home/xyj/PycharmProjects/flas... (26 minutes ago)
13:19 LF: UTF-8: Git: master:

图 8-5

（3）将本地 Python 解释器换成远程 Python 解释器。

在设置面板中，选择 Project:flask-demo 选项中的 Project Interpret，单击右侧齿轮，选择 Add Remote 添加远程解释器，在弹出的对话框中选择 Docker Compose，在 Server 中选择上一步新建的远程 API，Configuration file 是选择用于远程编译时候的配置，如图 8-6 所示。

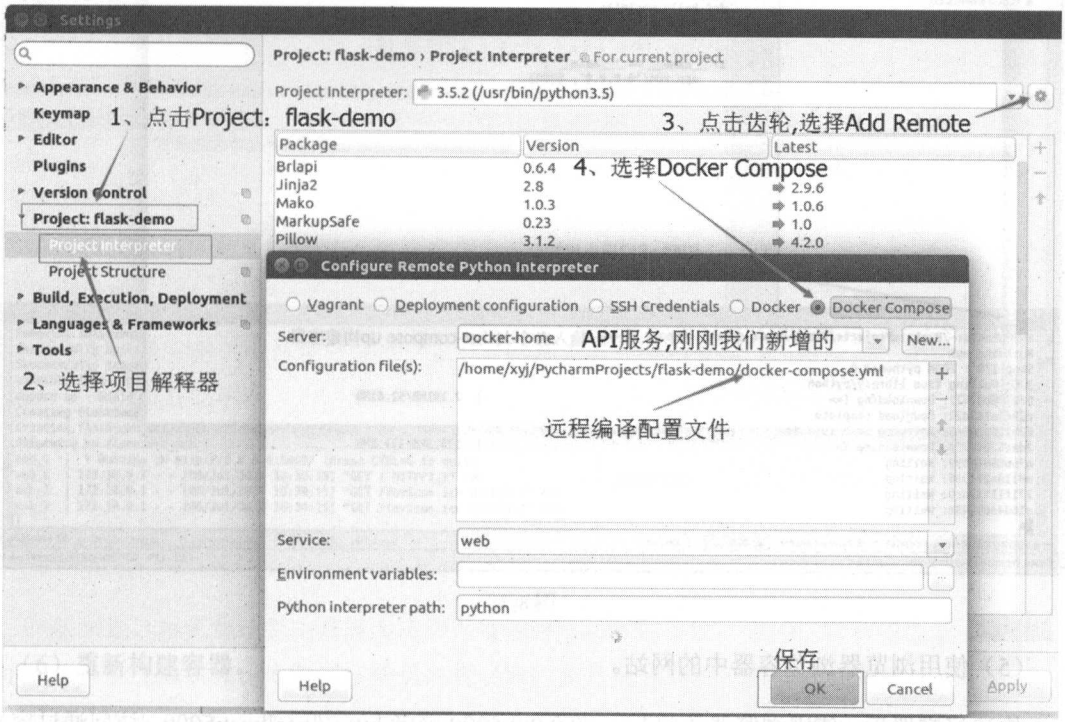


图 8-6

（4）在容器中运行代码。

配置好环境后，按快捷键 Alt+ F12 调出控制台，输入 Docker-compose up 命令可自动构建容器，并使用 Docker-compose.yml 中的配置启动容器，如图 8-7 所示。

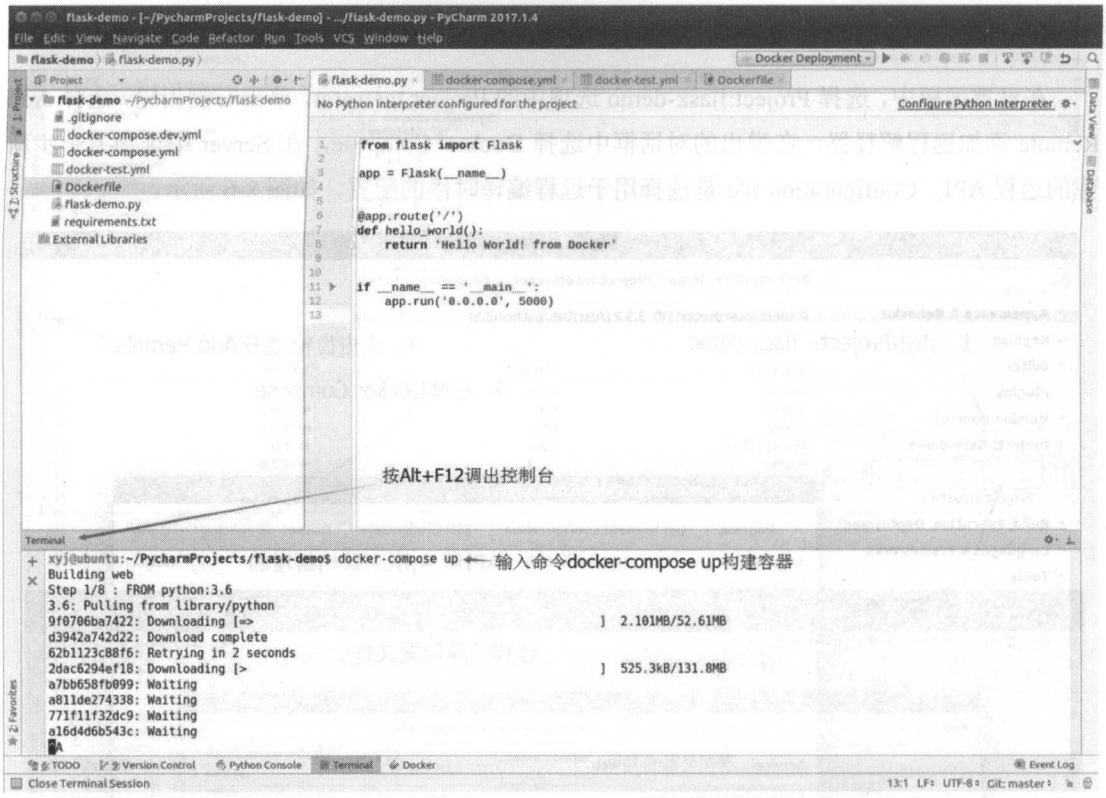


图 8-7

（5）使用浏览器浏览容器中的网站。

构建好容器后，使用浏览器访问 http://0.0.0.0:5000 或者 http://localhost:5000，访问通过容器生成的网站，如图 8-8 所示。

图 8-8

（6）重新构建容器。

当我们修改了项目的代码后仅可以通过 Docker-compose up --build 重新构建容器，如图 8-9 所示。

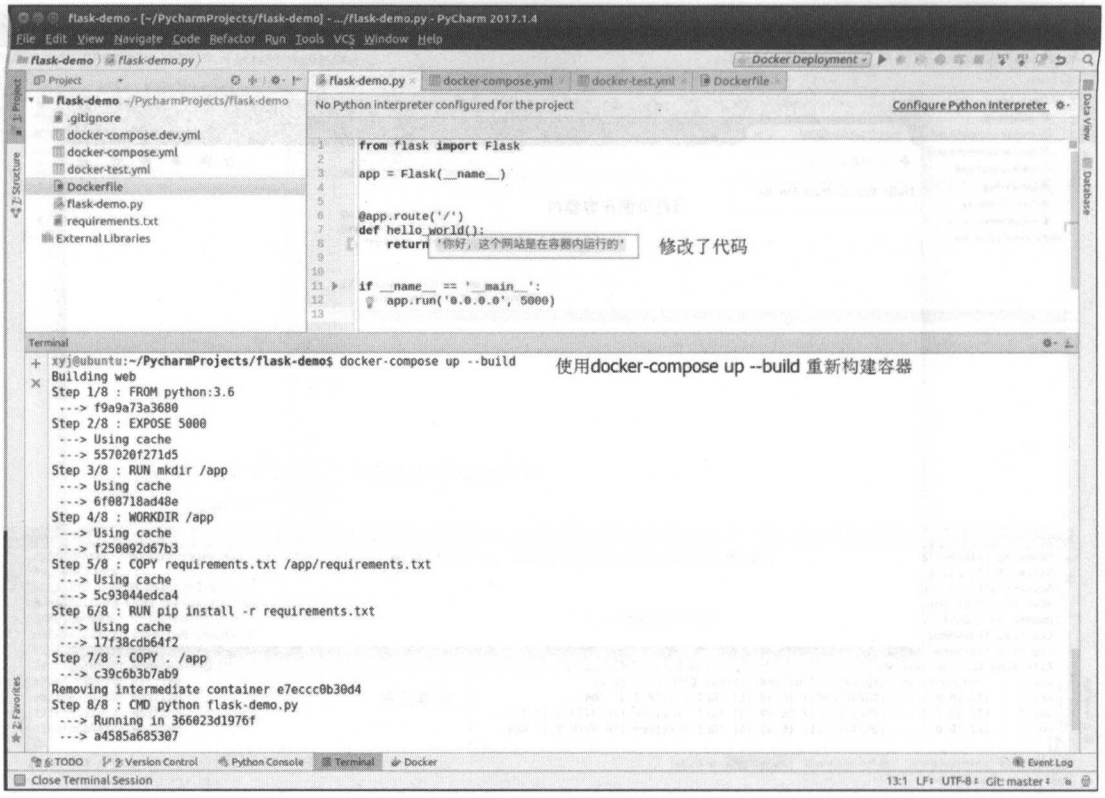

flask-demo - [~/PycharmProjects/flask-demo] - .../flask-demo.py - PyCharm 2017.1.4
File Edit View Navigate Code Refactor Run Tools VCS Window Help
flask-demo flask-demo.py
Docker Deployment
Project
flask-demo.py
docker-compose.yml
docker-test.yml
Dockerfile
flask-demo ~/PycharmProjects/flask-demo
.gitignore
docker-compose.dev.yml
docker-compose.yml
docker-test.yml
Dockerfile
flask-demo.py
requirements.txt
External Libraries
No Python interpreter configured for the project
Configure Python interpreter
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
return '你好，这个网站是在容器内运行的'
修改了代码
if __name__ == '__main__':
app.run('0.0.0.0', 5000)
Terminal
xyj@ubuntu:~/PycharmProjects/flask-demo$ docker-compose up --build
使用docker-compose up --build 重新构建容器
Building web
Step 1/8 : FROM python:3.6
---> f9a9a73a3680
Step 2/8 : EXPOSE 5000
---> Using cache
---> 557020f271d5
Step 3/8 : RUN mkdir /app
---> Using cache
---> 6f08718ad48e
Step 4/8 : WORKDIR /app
---> Using cache
---> f250092d67b3
Step 5/8 : COPY requirements.txt /app/requirements.txt
---> Using cache
---> 5c93044edca4
Step 6/8 : RUN pip install -r requirements.txt
---> Using cache
---> 17f38cdb64f2
Step 7/8 : COPY . /app
---> c39c6b3b7ab9
Removing intermediate container e7eccc0b30d4
Step 8/8 : CMD python flask-demo.py
---> Running in 366023d1976f
---> a4585a685307
6: TODO
9: Version Control
Python Console
Terminal
Docker
Event Log
Close Terminal Session
13:1 LF: UTF-8: Git: master:
1: Project
7: Structure
2: Favorites
Data View
Database

图 8-9

第9章
Web自动化测试

项目或代码做完之后，在交付使用部署之前，我们需要测试功能是否完备、是否正确。测试级别划分如下。

- ◎ 最小单元测试就是单元测试，即对代码基本单元进行测试，一般指对一个函数或一个方法进行测试；
- ◎ 在多个模块整合时需要集成测试；
- ◎ 当整合到系统的时候需要系统测试；
- ◎ 在完成交付使用的时候有验收测试；
- ◎ 在运行部署时发现有 BUG，修改代码后需要迭代测试或回归测试。

一般来说，单元测试是由开发人员边开发的时候边写的模块，直接在项目内创建 test 测试脚本即可。功能测试或压力测试这些是由测试部门在开发完成后写的测试项目。

1．单元测试

单元测试是指针对软件中最小单元（函数或方法）的正确性检验。在 Python 标准库中包含了单元测试组件 unittest，unittest 的灵感来源于 Java 中的 JUnit 单元测试框架。

通过继承 unittest.TestCase 类来自定义测试类，需要被测试的方法均以 test_开头，这样 unittest 才会把这个方法当成一个测试用例。运行完一个测试用例后，通过断言测试返回结果是否符合预期。在 unittest 中一般使用 self.assertIn()这样的方法来代替内置函数 assert 去断言，其好处是测试过程中 unittest 能收集所有测试结果，生成测试报告，如表 9-1 所示。

表 9-1

方　法	作　用	方　法	作用
assertEqual(a, b)	断言 a 等于 b	assertIsNone(x)	断言 x 为空
assertNotEqual(a, b)	断言 a 不等于 b	assertIsNotNone(x)	断言 x 不为空
assertTrue(x)	断言 x 为真	assertIn(a, b)	断言 a 在 b 内
assertFalse(x)	断言 x 为假	assertNotIn(a, b)	断言 a 不在 b 内
assertIs(a, b)	断言 a 是 b	assertIsInstance(a, b)	断言 a 为 b 类型
assertIsNot(a, b)	断言 a 不是 b	assertNotIsInstance(a, b)	断言 a 不是 b 类型

（1）Admin 模块简单测试脚本。

在 admin 模块下创建 test 目录，保存测试 admin 模块的测试用例脚本。

```
admin/test/test.py
#!/usr/bin/env python3
# encoding: utf-8

"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/12
"""
import os
import re
import blog_site
import unittest
import tempfile
import sqlite3


class MyTestCase(unittest.TestCase):
    def setUp(self):
        """ 测试开始前准备工作 """
        self.db_pathd, self.db_path = tempfile.mkstemp(suffix=".db")
# 创建临时数据文件
        # print(self.db_pathd, self.db_path)
        blog_site.app.config['SQLALCHEMY_DATABASE_URI'] =
'sqlite:///{}'.format(
            self.db_path.replace('\\', '/'))  # 测试后台数据库存放路径
        blog_site.app.config['TESTING'] = True  # 标记 TESTING 为真
```

```
        self.app = blog_site.app.test_client()  # 实例化测试客户端
        self.init_db(self.db_path)  # 初始化测试数据库

    def tearDown(self):
        """ 测试完成后清理工作 """
        os.close(self.db_pathd)  # 关闭数据文件句柄
        os.unlink(self.db_path)  # 删除数据文件

    def init_db(self, database):
        """ 初始化数据库往数据库内写数据 """
        db_path = database.replace('\\', '/')  # 转换 Windows 路径
        # print(db_path)
        db = sqlite3.connect(db_path)  # 连接到测试数据库
        c = db.cursor()  # 创建光标
        with open('creat_table.sql', 'r', encoding='utf-8') as f:
# 读取 SQL 脚本
            c.executescript(f.read())  # 批量执行 SQL 语句
        db.commit()  # 提交数据库保存

    def login(self, username, password):
        """ 登录功能 """
        return self.app.post('/admin/login/', data=dict(
            username=username,
            pwd=password
        ), follow_redirects=True)  # 使用 post 方法提交用户名和密码，参数
follow_redirects 为重定向追踪

    def test_login(self):
        """ 测试登录界面 """
        # 成功登录
        re_data = self.login('admin', '123456')
        print(re_data.data.decode())
        self.assertIn('欢迎 admin', re_data.data.decode())
# 断言'欢迎 admin'是否在返回结果中
        # 用户无权限
        re_data = self.login('test', '123456')
        self.assertIn('用户名或密码错误', re_data.data.decode())
# 断言'用户名或密码错误'是否在返回结果中
        # 用户名密码错误
        re_data = self.login('admin', '123')
        self.assertIn('用户名或密码错误', re_data.data.decode())
# 断言'用户名或密码错误'是否在返回结果中
```

```
    def logout(self):
        """ 登出 """
        return self.app.get('/admin/logout/', follow_redirects=True)
# get 访问退出路径

    def test_logout(self):
        """ 测试退出 """
        # 退出登录
        re_data = self.logout()
        self.assertIn('后台登录', re_data.data.decode())
        # 再次访问需授权页面
        re_data = self.app.get('/admin/user-list', follow_redirects=True)
        self.assertIn('请登录', re_data.data.decode())

    def test_add_user(self):
        """ 创建新用户 """
        # 管理员登录
        re_data = self.login('admin', '123456')
        self.assertIn('欢迎 admin', re_data.data.decode())
        # 访问用户列表页
        re_data = self.app.get('/admin/user-list/', follow_redirects=True)
        self.assertIn('test', re_data.data.decode())
        # get 访问新增用户
        re_data = self.app.get('/admin/user-list/edit/',
follow_redirects=True)
        token = re.findall(r'csrf_token.*value="(.*)"',
re_data.data.decode())  # 使用正则获取表单中的 csrf_token 值
        # post 提交新用户资料
        rv = self.app.post('/admin/user-list/edit/', data=dict(
            csrf_token='{}'.format(token[0]),
            name='test2',
            password='123456',
            role_id='1'
        ), follow_redirects=True)
        print(rv.data.decode())
        self.assertIn('test2', rv.data.decode())


if __name__ == '__main__':
    unittest.main()
```

（2）将多个测试用例整合成测试套件。

原则上每一个模块都定义了 test 测试脚本，因此多个模块就会有多个测试脚本，如果这些脚本不在同一个目录或同一个脚本内，那么运行测试的时候是或不是很麻烦？unittest 考虑到了这点，引入了测试套件概念，使用 TestSuite 将两个或多个以上测试用例聚合在一起执行。

在项目根目录中创建一个 run_test_suite.py 文件：

```
#!/usr/bin/env python3
# encoding: utf-8

"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/17
创建测试套件
"""
import unittest
from blog_site.admin.test.test import TestAdminCase
# 引入 admin 模块测试用例
from blog_site.blog.test.test import TestBlogCase
# 引入 blog 模块测试用例
admin_test = unittest.TestLoader().loadTestsFromTestCase(TestAdminCase)
# 载入 admin 模块测试用例
blog_test = unittest.TestLoader().loadTestsFromTestCase(TestBlogCase)
# 载入 blog 模块测试用例

suite = unittest.TestSuite([admin_test, blog_test])  # 定义测试套装

if __name__ == '__main__':
    runner = unittest.TextTestRunner()  # 实例化文本测试运行器
    runner.run(suite)  # 运行测试器
```

2. 功能测试

功能测试就是对产品的各个功能进行验证，根据功能测试用例逐项测试，检查产品是否达到要求的功能。[1] 网站项目开发完成后，一般会使用一些 Web 应用程序测试工具对其各个功能

1　功能测试解释引用
http://baike.baidu.com/link?url=kPlBsUIsvG-otR-AvPo3xa1kK_JcwSTqDm1fXT4qdAoKqd03lXYYFu4j6QpAf6aLQVRU60FZIa96YumaZx99hAWeW9wFY4vdwywfbhHEROB9sKaZHji9GujzmgGSNmhq

进行测试，例如使用 Seleium 框架。

创建新项目用于测试第 17 章中的个人博客后台。

（1）查看使用的 Chrome 版本，如图 9-1 所示。

图 9-1

（2）查看 Chrome 驱动更新日志。

Chrome 驱动主页为 https://sites.google.com/a/chromium.org/chromedriver/home。

打开网址 http://chromedriver.storage.googleapis.com/2.30/notes.txt，查看最新的 Chrome 驱动更新日志，如图 9-2 所示。

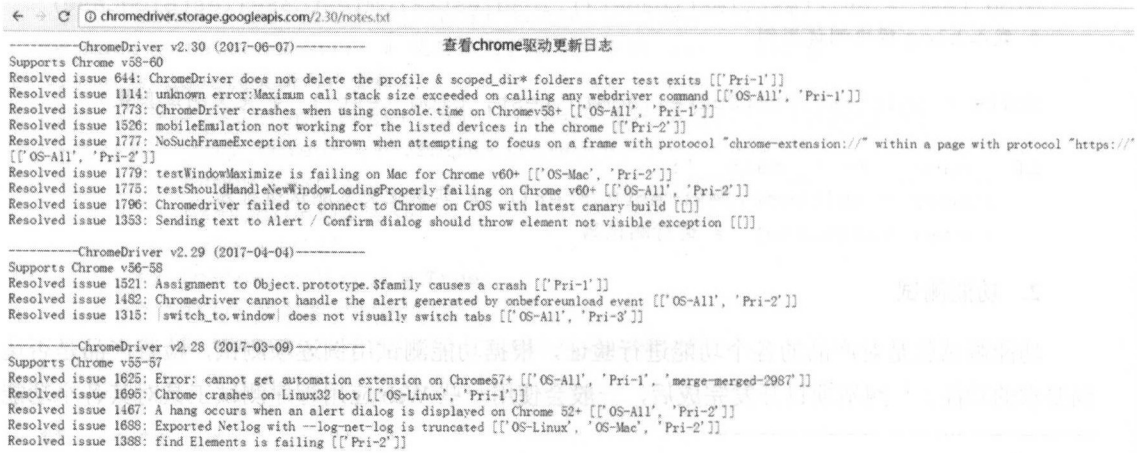


图 9-2

（3）下载 Chrome 驱动。

打开网址 http://chromedriver.storage.googleapis.com/index.html?path=2.30/，下载开发系统对应的版本驱动，如图 9-3 所示。

图 9-3

（4）测试创建项目结构。

使用 Page Object 设计模式创建测试项目，使用 Page Object 的好处是可以将页面元素和业务逻辑分开编写，当网页改版的时候，只需修改相应的页面元素定位即可，不需要修改测试逻辑。具体结构如图 9-4 所示。

```
E:.
    .gitignore  # git忽略文件
    element.py  # 页面元素
    HTMLTestRunner.py  # 测试报告生成模块
    locators.py  # 元素定位
    page.py  # 页面对象类
    requirements.txt  # 依赖包
    test_case.py  # 测试用例
    __init__.py  # 包引导

├─browser_drive  # 存放浏览器驱动
        chromedriver.exe  # chrome v58 驱动

├─report  # 存放测试报告
        TestReport_2017-07-19 00_05_28.html  # 生成测试报告
```

图 9-4

PyCharm 中的项目如图 9-5 所示。

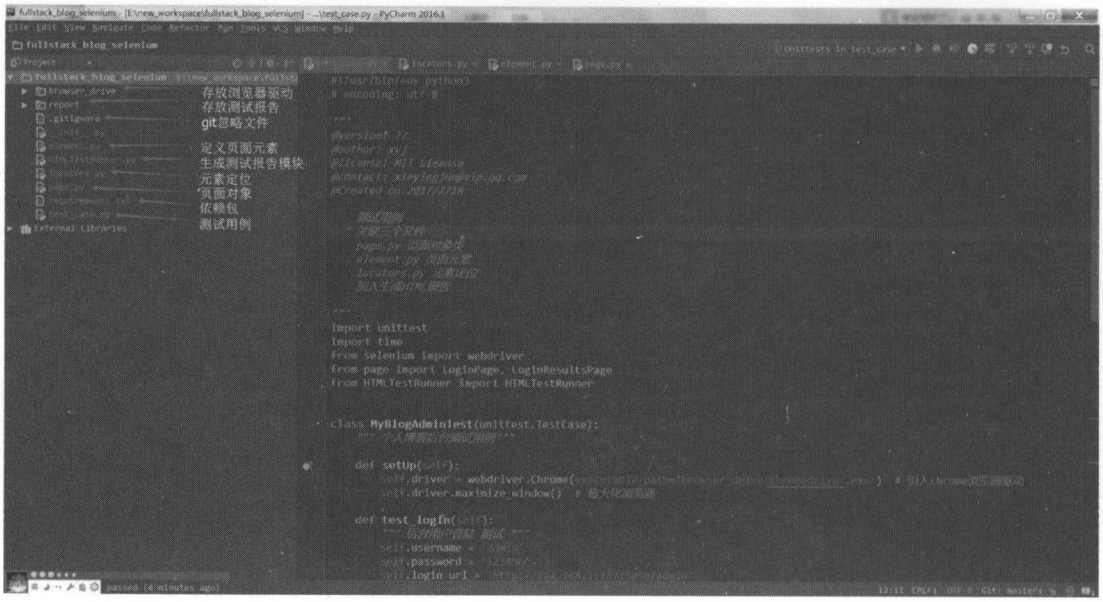


图 9-5

（5）定义页面元素。

element.py

```
#!/usr/bin/env python3
# encoding: utf-8

"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/18

定义页面元素

"""
from selenium.webdriver.support.ui import WebDriverWait


class NamePageElement(object):
    """ 通过 Name 查找元素 """
```

```
    def __set__(self, obj, value):
        """ 将文本内容发送到指定对象中 """
        driver = obj.driver
        WebDriverWait(driver, 100).until(
            lambda driver: driver.find_element_by_name(self.locator))
        driver.find_element_by_name(self.locator).send_keys(value)

    def __get__(self, obj, owner):
        """ 获取指定对象的文本内容 """
        driver = obj.driver
        WebDriverWait(driver, 100).until(
            lambda driver: driver.find_element_by_name(self.locator))
        element = driver.find_element_by_name(self.locator)
        return element.get_attribute("value")


class IdPageElement(object):
    """ 基于 ID 查找元素 """

    def __set__(self, obj, value):
        """ 将文本内容发送到指定对象中 """
        driver = obj.driver
        WebDriverWait(driver, 100).until(
            lambda driver: driver.find_element_by_id(self.locator))
        driver.find_element_by_id(self.locator).send_keys(value)

    def __get__(self, obj, owner):
        """ 获取指定对象的文本内容 """
        driver = obj.driver
        WebDriverWait(driver, 100).until(
            lambda driver: driver.find_element_by_id(self.locator))
        element = driver.find_element_by_id(self.locator)
        return element.get_attribute("value")
```

（6）定位页面元素。

```
locators.py:
#!/usr/bin/env python3
# encoding: utf-8

"""
@version: ??
@author: xyj
```

```
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/18

定义元素定位

"""
from selenium.webdriver.common.by import By


class Login(object):
    """ 登录页登录按钮 """
    LOGIN_BUTTON = (By.XPATH, '//button[@type="submit"]')  # 通过XPATH定位
#登录按钮


class GoIndex(object):
    """ 返回首页按钮 """
    GO_INDEX_BUTTON = (By.ID, 'submit')
```

(7) 定义页面对象类。

```
Page.py;
#!/usr/bin/env python3
# encoding: utf-8

"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/18

定义页面对象类

"""

from locators import Login, GoIndex
from element import NamePageElement, IdPageElement


class LoginName(NamePageElement):
    """ 定位用户名输入框 """
    locator = 'username'
```

```
class LoginPassWord(IdPageElement):
    """定位密码输入框"""
    locator = 'pwd'


class BasePage:
    """ 定义基页 """

    def __init__(self, driver):
        self.driver = driver


class LoginPage(BasePage):
    login_name = LoginName()
    login_password = LoginPassWord()

    def is_title_matches(self):
        """ 验证标题是否为"管理员登录" """
        return "管理员登录" in self.driver.title

    def click_go_login_button(self):
        """ 点击登录按钮 """
        element = self.driver.find_element(*Login.LOGIN_BUTTON)
        element.click()

    def click_go_button(self):
        """ 点击返回主页按钮 """
        element = self.driver.find_element(*GoIndex.GO_INDEX_BUTTON)
        element.click()


class LoginResultsPage(BasePage):
    """ 验证登录后返回页是否正常 """

    def is_results_found(self):
        return "欢迎 admin" in self.driver.page_source
```

（8）编写测试脚本。

```
test_case.py:
#!/usr/bin/env python3
# encoding: utf-8
```

```
"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/18

    测试用例
    关联三个文件
    page.py 页面对象类
    element.py 页面元素
    locators.py 元素定位
    加入生成 HTML 报告

"""
import unittest
import time
from selenium import webdriver
from page import LoginPage, LoginResultsPage
from HTMLTestRunner import HTMLTestRunner


class MyBlogAdminTest(unittest.TestCase):
    """ 个人博客后台测试用例"""

    def setUp(self):
        self.driver =
webdriver.Chrome(executable_path='browser_drive/chromedriver.exe')
# 引入 chrome 浏览器驱动
        self.driver.maximize_window()  # 最大化浏览器

    def test_login(self):
        """ 后台用户登录 测试 """
        self.username = 'admin'
        self.password = '123456'
        self.login_url = 'http://192.168.1.137:5050/admin'
        self.driver.get(self.login_url)

        login_page = LoginPage(self.driver)  # 实例化主页
        self.assertTrue(login_page.is_title_matches)  # 断言页面标题是否正确
        login_page.login_name = self.username  # 设置登录名
        login_page.login_password = self.password  # 设置登录密码
```

```
        login_page.click_go_login_button()  # 点击登录按钮

        login_results_page = LoginResultsPage(self.driver)  # 登录结果
        self.assertTrue(login_results_page.is_results_found())

    def tearDown(self):
        self.driver.close()


def getNowTime():
    """
    获取当前系统时间
    """
    return time.strftime("%Y-%m-%d %H%M%S", time.localtime(time.time()))


def runAutomation():
    """ 生成测试报告 """
    suite = unittest.makeSuite(MyBlogAdminTest)  # 生成测试套件
    filename = 'report/' + 'TestReport_' + getNowTime() + '.html'
# 测试报告名
    fp = open(filename, 'wb')  # 创建文件
    runner = HTMLTestRunner(stream=fp, title='自动化测试报告', description='
自动化测试报告详细的信息')  # 生成测试报告
    runner.run(suite)  # 运行测试套件
    fp.close()  # 关闭文件


if __name__ == "__main__":
    # unittest.main()
    runAutomation()
```

（9）测试结果如图 9-6 所示。

自动化测试报告

Start Time: 2017-07-19 00:05:28
Duration: 0:00:07.690440
Status: Pass 1

自动化测试报告详细的信息

Show Summary Failed All

Test Group/Test case	Count	Pass	Fail	Error	View
MyBlogAdminTest: 个人博客后台测试用例	1	1	0	0	Detail
test_login: 后台用户登陆 测试			pass		
Total	1	1	0	0	

图 9-6

整个项目的详细代码均可从https://gitlab.com/lanmaokafei/fullstack_blog_selenium下载使用。

3. 测试编写用例

一般来说，项目开发完成后会交由测试部进行测试，而测试部门在测试一个项目的时候会安排不同的人员来测试，因而测试人员之间的交流需要通过书面的文档来指导测试流程，这份文档就是测试用例，如图 9-7 所示。测试用例要写得明晰，能让不懂测试的人员拿到手后知道如何去做手工测试。

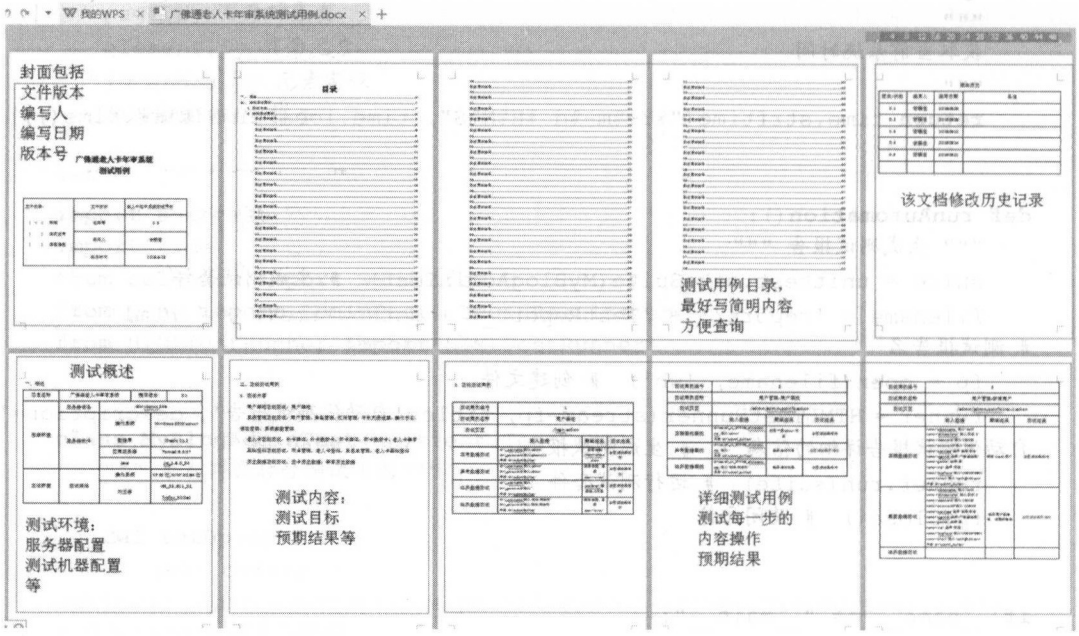


图 9-7

测试用例文档如表 9-2 所示。

表 9-2

测试用例编号	1		
测试用例名称	后台用户登录		
测试页面	/admin/login/		
	输入数据	期望结果	测试结果
正常数据测试	Id='username'用户名输入框输入“admin” Id='pwd'密码输入框输入‘123456’ type="submit"单击登录按钮	页面转跳/admin/	与预期结果相符

续表

异常数据测试	Id='username'用户名输入框输入‘admin’ Id='pwd'密码输入框输入‘1234’ type="submit"单击登录按钮	登录失败，显示错误 用户名或密码错误	与预期结果相符
非授权用户测试	Id='username'用户名输入框输入‘test’ Id='pwd'密码输入框输入‘123456’ type="submit"单击登录按钮	登录失败，显示错误 用户名或密码错误	与预期结果相符
边界数据测试	Id='username'用户名输入框输入 Id='pwd'密码输入框输入 type="submit"单击登录按钮	登录失败，显示错误 用户名或密码错误	与预期结果相符
边界数据测试	Id='j_username'输入“填充满字符” Id='j_password'输入“填充满字符” 单击 id="submitButton"	登录失败，显示错误 用户名或密码错误	与预期结果相符

一个模块的功能应尽可能覆盖数据的测试，这样在部署的时候漏洞就相对较少。根据测试用例写成自动化测试脚本，这样开发完成或修改代码后可进行一下回归测试，这就涉及持续集成了。

第10章
持续集成

GitLab-runner是一个配合gitlab ci一起使用的后台运行任务，并把结果返给GitLab的服务。GitLab-runner允许使用令牌方式运行多个任务，可以工作在多种模式下，包括本机、使用Docker容器、在容器内使用 SSH 执行任务、在容器内连接不同云或虚拟化程序进行自动缩放、通过SSH连接到远程服务器等。

（1）下载GitLab-runner。

GitLab-runner能运行在多种环境下，这里演示直接使用Docker容器运行。运行镜像的前提是需要安装好Docker服务。使用Docker pull命令下载由GitLab官方提供的GitLab-runner镜像，如图10-1所示。

```
$ docker pull gitlab/gitlab-runner:latest  # 使用 docker pull 拉取最新镜像
```

```
xyj@xyj-home:~$ docker pull gitlab/gitlab-runner
Using default tag: latest
latest: Pulling from gitlab/gitlab-runner
Digest: sha256:ffef40a0f8a622f4dc08f24642f68087e34929d08455ec07055105154c1d9e38
Status: Image is up to date for gitlab/gitlab-runner:latest
xyj@xyj-home:~$
```

使用docker pull gitlab/gitlab-runner下载由gitlab官方提供的gitlab-runner最新镜像

图10-1

（2）启动GitLab-runner服务。

先在宿主机上创建用于保存GitLab-runner配置的目录，运行GitLab-runner，如图10-2所示。

```
$ mkdir gitlab-ci_volume  # 创建用于保存 GitLab-runner 配置的目录
$ docker run -d \  # 后台运行
--name c-gitlab-ci \  # 别名
-v //home/xyj/gitlab-ci_volume/config:/etc/gitlab-runner \
# 将本地磁盘映射到容器内，用于保存 GitLab-runner 配置
```

```
    -v /var/run/docker.sock:/var/run/Docker.sock \
    # 使用宿主机的 docker.sock 控制 docker
    gitlab/gitlab-runner  # 运行镜像
```

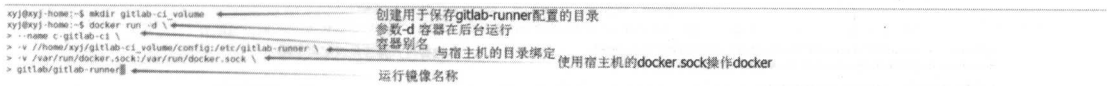


图 10-2

（3）在 GitLab 项目中查找持续集成地址和令牌。

打开 GitLab 页面上的项目，这里是 http://192.168.1.137:8000/xyj/flask-demo，单击 Settings 进入项目设置，在次级目录中选择 CI/CD Pipelines，记录 Specific Runners 中提供的 URL 和 token 以备后续使用，如图 10-3 和图 10-4 所示。

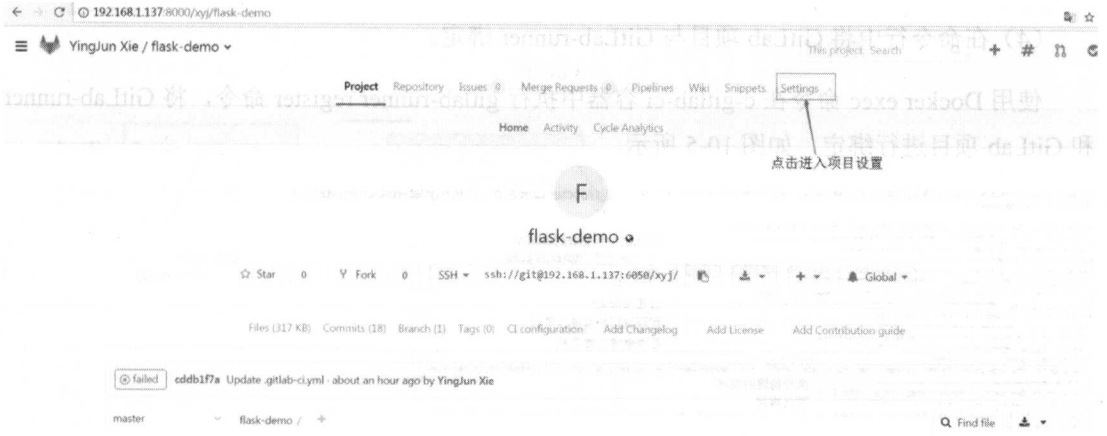


图 10-3

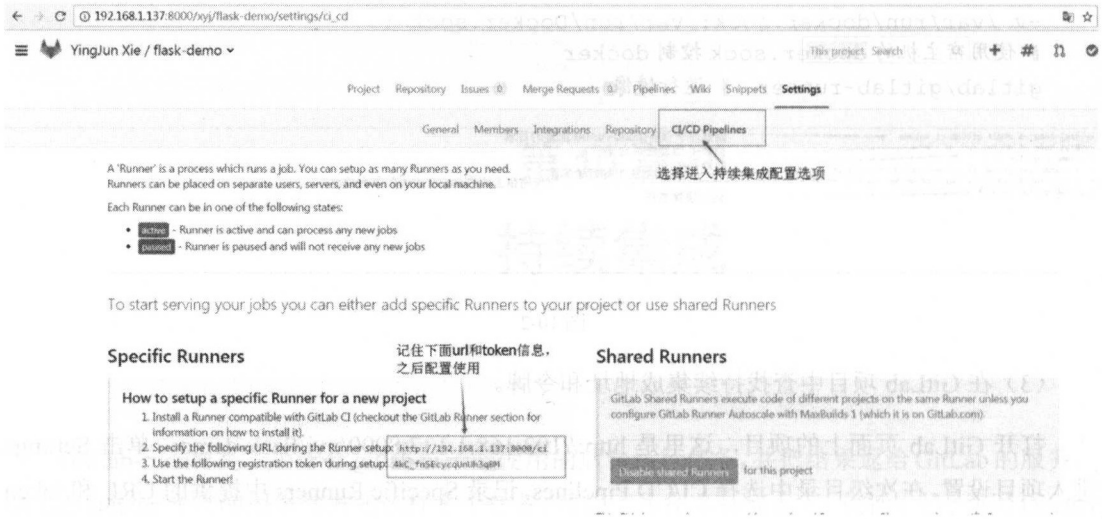


图 10-4

（4）在命令行中将 GitLab 项目与 GitLab-runner 绑定。

使用 Docker exec 命令在 c-gitlab-ci 容器中执行 gitlab-runner register 命令，将 GitLab-runner 和 GitLab 项目进行绑定，如图 10-5 所示。

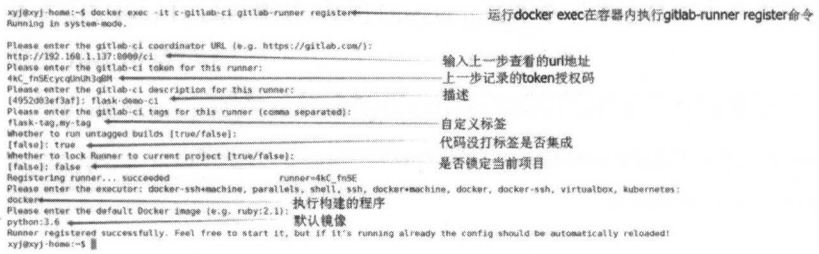


图 10-5

（5）创建.gitlab-ci.yml 文件。

GitLab 7.12 版本开始使用.gitlab-ci.yml 文件来控制 GitLab-runner 自动构建的配置文件，.gitlab-ci.yml 文件存放在项目的根目录下。浏览项目主页，单击 Set up CI 按钮进入自动构建配置页面，如图 10-6 和图 10-7 所示。

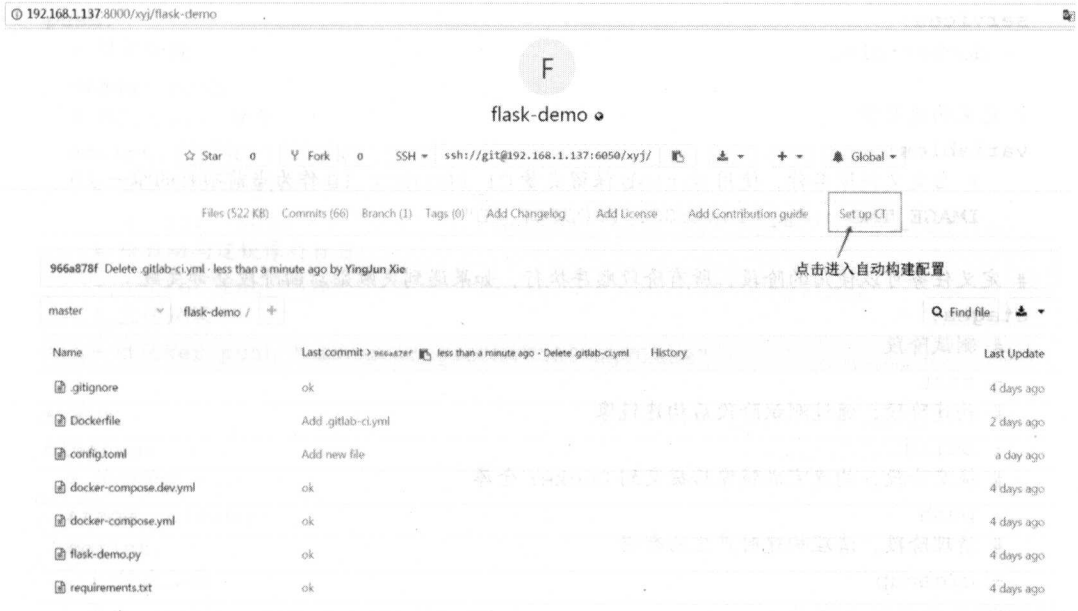


图 10-6

图 10-7

（6）配置.gitlab-ci.yml 文件。

```
# 使用 Docker 中的 Docker 镜像
# image: python:3.6
image: docker:latest

# 服务定义了在构建期间运行的另一个 Docker 镜像
```

```
services:
 - docker:dind

# 定义构建变量
variables:
    # 自定义镜像名称，使用 GitLab 保留变量 CI_PROJECT_ID 作为当前项目的唯一 ID
    IMAGE_TAG : "myflask:$CI_PROJECT_ID"

# 定义任务可以使用的阶段，所有阶段顺序执行，如果遇到失败则后面阶段全部失败
stages:
  # 测试阶段
  - test
  # 构建阶段，通过测试阶段后构建镜像
  - build
  # 提交阶段，构建完成镜像后提交到 Docker 仓库
  - push
  # 清理阶段，清理构建时产生的数据
  - cleanup

# 任务名，可以自定义
build:

  # 该任务所在阶段
  stage: build
  # 执行 shell 命令
  script:
#    - docker info
    - docker build  -t "$IMAGE_TAG" .

test:
  image: python:3.6
  stage: test
  script:
    - export PYTHONPATH=/builds/xyj/fullstack_blog
    - apt-get update -qy
    - apt-get install -y python3-dev python3-pip
    - pip install -r requirements.txt
    - cd blog_site/admin/test/
    - echo $PYTHONPATH
    # - pwd
    - python test.py

# 提交
```

```
push:
  # 提交阶段
  stage: push
  # 执行 shell 命令
  script:
    # 登录本地 registry
    - docker login -u "xyj" -p "docker-registry" myregistry.com:8080
    # 给自动构建镜像打标签
    - docker tag "$IMAGE_TAG" "myregistry.com:8080/myflask"
    # 上传镜像
    - docker push "myregistry.com:8080/myflask"

# 清理
cleanup:
  # 清理阶段
  stage: cleanup
  script:
    # 停止容器
    # - docker stop c-test-flask
    # 移除容器
    # - docker rm c-test-flask
    # 移除生成的镜像
    - docker rmi "$IMAGE_TAG"
```

（7）查看构建任务消息，如图 10-8 所示。

图 10-8

（8）填坑：解决 Cannot connect to the Docker daemon。

在提交集成后，查看日志集成故障，提示如图 10-9 所示。

图 10-9

解决方法：

与容器交互编辑 docker-runner 的配置文件 config.toml。

```
$ docker exec -it c-gitlab-ci vi /etc/gitlab-runner/config.toml
# 使用 vi 编辑 config.toml
# 将 volumes = ["/cache"] 更改成 volumes =
# ["/var/run/docker.sock:/var/run/docker.sock", "/cache"]，即把宿主机的
# docker.sock 映射到 docker-runner 中，如图 10-10 所示。
```

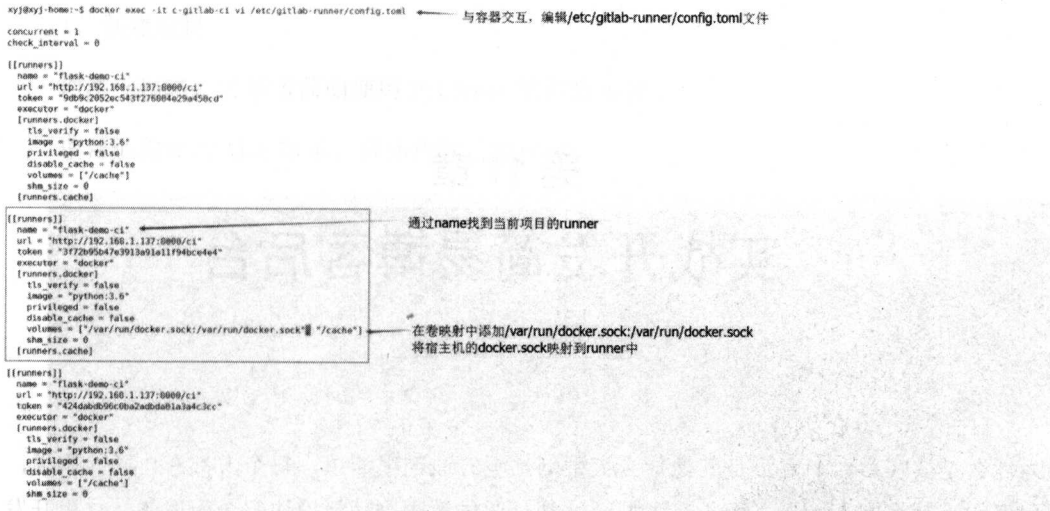

xyj@xyj-home:~$ docker exec -it c-gitlab-ci vi /etc/gitlab-runner/config.toml
与容器交互，编辑/etc/gitlab-runner/config.toml文件
concurrent = 1
check_interval = 0
[[runners]]
name = "flask-demo-ci"
url = "http://192.168.1.137:8000/ci"
token = "9db9c2052ec543f276084e29a450cd"
executor = "docker"
[runners.docker]
tls_verify = false
image = "python:3.6"
privileged = false
disable_cache = false
volumes = ["/cache"]
shm_size = 0
[runners.cache]
[[runners]]
name = "flask-demo-ci"
通过name找到当前项目的runner
url = "http://192.168.1.137:8000/ci"
token = "3f72b95b47e3913a91a11f94bce4e4"
executor = "docker"
[runners.docker]
tls_verify = false
image = "python:3.6"
privileged = false
disable_cache = false
volumes = ["/var/run/docker.sock:/var/run/docker.sock", "/cache"]
在卷映射中添加/var/run/docker.sock:/var/run/docker.sock
将宿主机的docker.sock映射到runner中
shm_size = 0
[runners.cache]
[[runners]]
name = "flask-demo-ci"
url = "http://192.168.1.137:8000/ci"
token = "424dabdb96c0ba2adbdab1a3a4c3cc"
executor = "docker"
[runners.docker]
tls_verify = false
image = "python:3.6"
privileged = false
disable_cache = false
volumes = ["/cache"]
shm_size = 0

图 10-10

第 11 章

实战开发简易博客后台

为什么是博客后台，而不是其他什么大型网站呢？原因很简单，每个人都有自己的需求，很多深度的定制都因人而异，而所有大型网站都是通过不同小功能模块组合而成的，这里开发个人博客后台简单又实用。说白了就是围绕着数据库增删改查四个操作，按照不同的需求实现不同的输出。

前面很多案例都是直接用 flask 框架写的代码来运行网站，为什么本书会选择 flask 框架而不是 Django 这类大型框架呢？

Django 框架是非常优秀的框架，是目前市场上 Web 应用或招聘 Web 开发的主流框架。Django 的特点是 Web 开发中所需的各个方面功能都考虑到了，比如说关于状态管理的 Session 或 Cookie，关于数据库操作的 ORM 映射、请求响应模板都做得很完善，在使用 Django 开发 Web 网站的时候往往会感觉是使用一门叫作“Django”的语言在开发，没 Python 语言什么事。Django 的优点是 Web 开发应用的所有方向都有整套解决方案实现，缺点是只能用 Django 提供的解决方案实现，灵活性相对较低。

Flask 框架刚好弥补了 Django 框架的缺点，Flask 被称为微框架，所谓微框架是核心比较精简，默认情况下，Flask 不包含数据库抽象层、表单验证或者其他已有的库可以处理的东西，但这并不意味着 Flask 框架不能开发大型 Web 应用。Flask 核心简洁的目的是不限制开发功能过程中选择的解决方案，例如状态管理、数据库操作和请求响应这些，开发人员可以根据自己的需求选择不同的解决方案，其灵活性大大高于 Django 框架。

下面进入正题，省略了创建项目和使用 Git 跟踪项目步骤，具体过程可参考前面章节。

1. 创建项目

创建过程略，可参考前面使用 PyCharm 的创建步骤。

项目结构如图 11-1 所示，具体代码后面详述。

图 11-1

Blog_stie/__init__.py

```
#!/usr/bin/env python3
# encoding: utf-8
"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/11
"""
from flask import Flask, redirect, url_for
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager
app = Flask(__name__)
app.config.from_object('conf')  # 从 conf.py 中加载配置信息
db = SQLAlchemy(app)  # 构造数据库核心对象实例，并初始化程序实例
login_manager = LoginManager()  # 创建登录类
login_manager.init_app(app)  # 和 Flask-Login 协同工作的代码
```

```
login_manager.login_view = 'admin.login'
# 当未登录的用户尝试访问一个 login_required
# 装饰的视图时，Flask-Login 会闪现一条消息并且重定向到登录视图。与 url_for() 用法相同
login_manager.login_message = '请登录'  # 自定义未登录消息提示
login_manager.login_message_category = '错误'  # 定义未登录消息提示的分类

def register_blueprints():
    """ 注册不同模块蓝图 """
    # 注册管理后台模块
    from blog_site.admin.views import admin
    from blog_site.admin import models
    app.register_blueprint(admin, url_prefix='/admin')
    # 注册博客模块
    from blog_site.blog import models
    from blog_site.blog.views import blog
    app.register_blueprint(blog, url_prefix='/blog')

@app.route('/')
def index():
    return redirect(url_for('blog.index'))

register_blueprints()  # 注册蓝图
```

2. 管理数据模型

主要记录用户权限和用户信息，保存可以登录后台的信息，如图 11-2 所示。

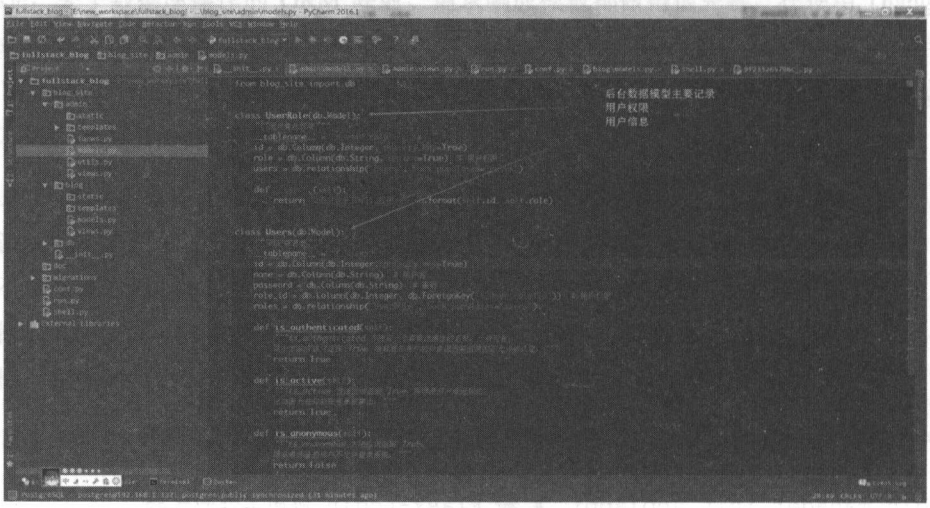


图 11-2

```
Admin/models.py
#!/usr/bin/env python3
# encoding: utf-8
"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/11
"""
from blog_site import db

class UserRole(db.Model):
    """用户角色权限"""
    __tablename__ = 't_user_role'
    id = db.Column(db.Integer, primary_key=True)
    role = db.Column(db.String, unique=True)  # 用户权限
    users = db.relationship('Users', back_populates='roles')
    def __repr__(self):
        return '<用户角色 ID:{},权限:{} >'.format(self.id, self.role)

class Users(db.Model):
    """用户信息表"""
    __tablename__ = 't_users'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)  # 用户名
    password = db.Column(db.String)  # 密码
    role_id = db.Column(db.Integer, db.ForeignKey('t_user_role.id'))
# 用户权限
    roles = db.relationship('UserRole', back_populates='users')
    def is_authenticated(self):
        """is_authenticated 方法有一个具有迷惑性的名称。一般而言，这个方法应该只返
        回 True，除非表示用户的对象因为某些原因不允许被认证。"""
        return True
    def is_active(self):
        """is_active 方法应该返回 True，除非用户是无效的，比如他们的账号是被禁止的。
        """
        return True
    def is_anonymous(self):
        """is_anonymous 方法应该返回 True，除非是伪造的用户不允许登录系统。"""
        return False
    def get_id(self):
```

```
        """get_id 方法应该返回一个用户唯一标识符，以 unicode 格式。
        我们使用数据库生成的唯一 id。"""
        try:
            return str(self.id)
        except Exception as e:
            print(e)
    def __repr__(self):
        return '<用户名字:{},权限:{} >'.format(self.name, self.role_id)
```

博客数据模型主要记录日志类型、日志内容和留言信息，前端博客数据模型如图 11-3 所示。

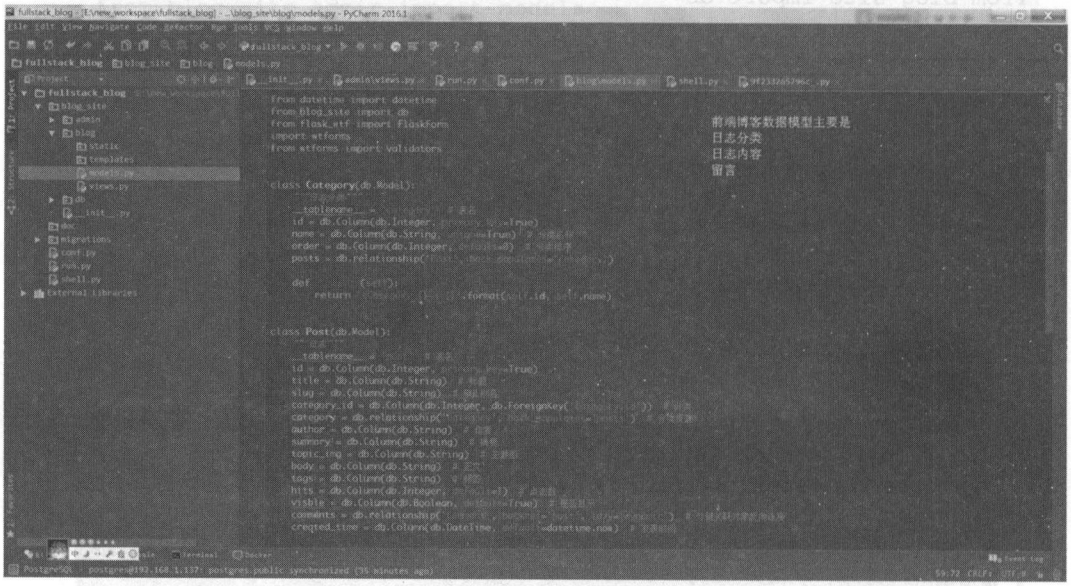


图 11-3

```
Blog/models.py
#!/usr/bin/env python3
# encoding: utf-8
"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/11
"""
from datetime import datetime
from blog_site import db
```

```
from flask_wtf import FlaskForm
import wtforms
from wtforms import validators

class Category(db.Model):
    """日志分类"""
    __tablename__ = 'category'  # 表名
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String, unique=True)  # 分类名称
    order = db.Column(db.Integer, default=0)  # 分类排序
    posts = db.relationship('Post', back_populates='category')
    def __repr__(self):
        return '<Category {} : {}'.format(self.id, self.name)

class Post(db.Model):
    """日志"""
    __tablename__ = 'post'  # 表名
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String)  # 标题
    slug = db.Column(db.String)  # RUL 别名
    category_id = db.Column(db.Integer, db.ForeignKey('category.id'))
# 分类
    category = db.relationship('Category', back_populates='posts')
# 分类反查
    author = db.Column(db.String)  # 作者
    summary = db.Column(db.String)  # 摘要
    topic_img = db.Column(db.String)  # 主题图
    body = db.Column(db.String)  # 正文
    tags = db.Column(db.String)  # 标签
    hits = db.Column(db.Integer, default=1)  # 点击数
    visble = db.Column(db.Boolean, default=True)  # 是否显示
    comments = db.relationship('Comment', backref='post', lazy='dynamic')
# 外键关联对象反向连接
    created_time = db.Column(db.DateTime, default=datetime.now)  # 发表时间
    def __repr__(self):
        return '<Post {}:{}>'.format(self.id, self.title)

class Comment(db.Model):
    """ 留言 """
    __tablename__ = 'comments'
    id = db.Column(db.Integer, primary_key=True)
```

```
    name = db.Column(db.String(255))  # 留言用户
    text = db.Column(db.Text())  # 留言内容
    date = db.Column(db.DateTime())  # 留言时间
    post_id = db.Column(db.Integer, db.ForeignKey('post.id'))  # 反查日志 id
    def __repr__(self):
        return "<Comment '{}'>".format(self.name)

class PostForm(FlaskForm):
    title = wtforms.StringField('标题',
validators=[validators.DataRequired('标题必填!')])
    slug = wtforms.StringField('RUL 别名')
    category_id = wtforms.SelectField('分类')
    author = wtforms.StringField('作者')
    summary = wtforms.TextAreaField('摘要')
    topic_img = wtforms.FileField('主题图')
    body = wtforms.TextAreaField('正文')
    tags = wtforms.StringField('标签')
    hits = wtforms.IntegerField('点击数', default=1)
    visible = wtforms.BooleanField('显示', default=True)
    created_time = wtforms.DateTimeField('发表时间')

class CategoryForm(FlaskForm):
    name = wtforms.StringField('分类名称',
validators=[validators.DataRequired('内容必填!')])
    order = wtforms.IntegerField('分类排序')

class CommentForm(FlaskForm):
    name = wtforms.StringField('留言用户',
validators=[validators.DataRequired('内容必填!')])
    text = wtforms.TextAreaField('留言内容')
```

3. 创建数据库，使用 flask-migrate 进行跟踪

首先使用 flask-migrate 初始化项目数据库跟踪，这时会在项目根目录下生成一个 migrations 文件，用于保存数据库结构差异记录和迁移脚本，如图 11-4 所示。

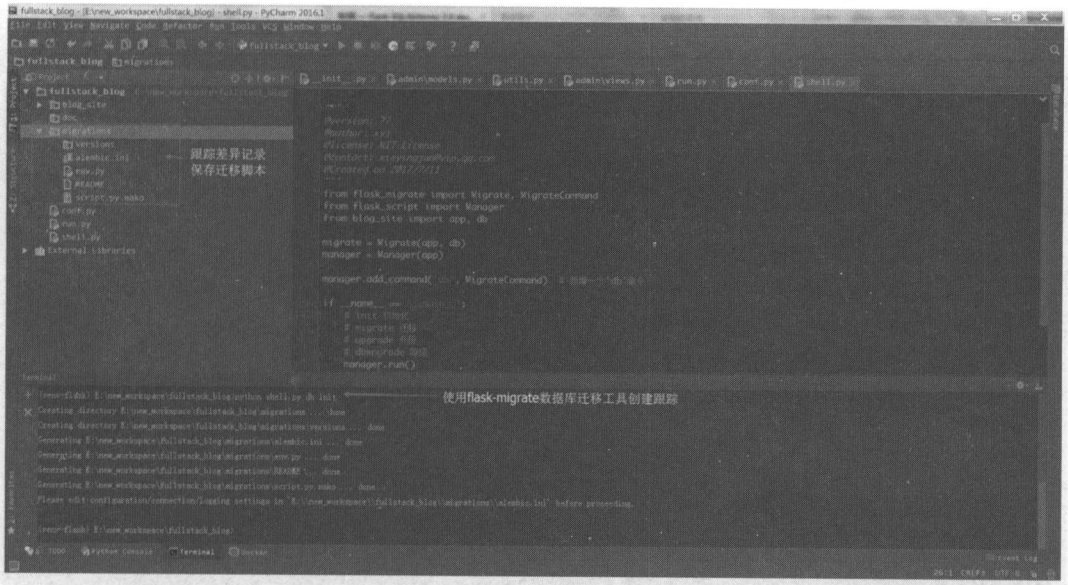


图 11-4

```
>python shell.py db init  # 初始化数据库跟踪
Shell.py
#!/usr/bin/env python3
# encoding: utf-8
"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/11
"""
from flask_migrate import Migrate, MigrateCommand
from flask_script import Manager
from blog_site import app, db
migrate = Migrate(app, db)
manager = Manager(app)
manager.add_command('db', MigrateCommand)  # 新增一个'db'命令
if __name__ == '__main__':
    # init 初始化
    # migrate 迁移
    # upgrade 升级
    # downgrade 降级
    manager.run()
```

然后创建数据表，操作如图 11-5 所示。

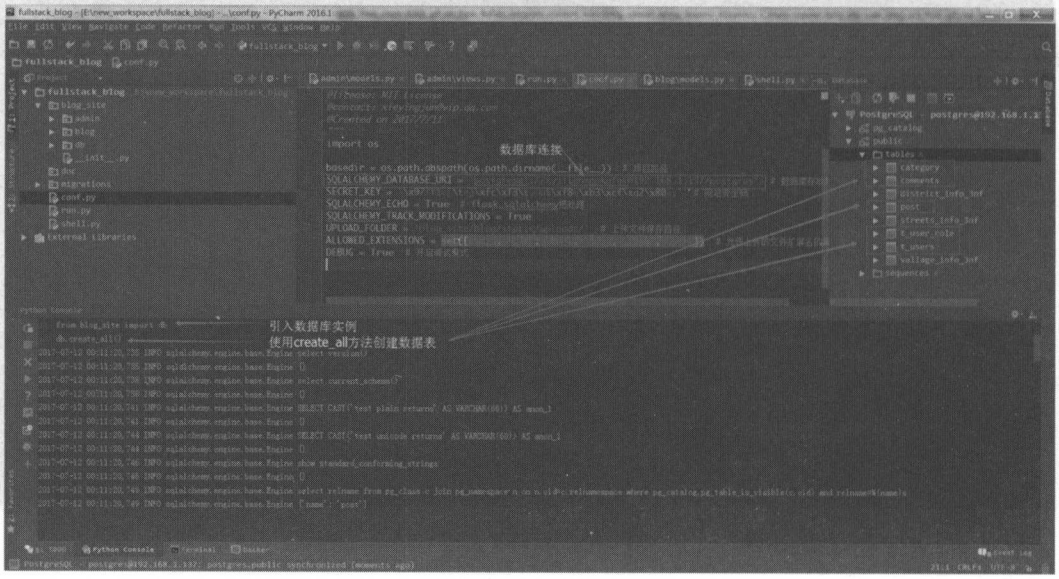


图 11-5

```
>>> from blog_site import db  # 引入数据库实例
>>> db.create_all()  # 创建所有表
```

再次使用 flask-migrate 跟踪数据变化，对比上一次数据表结构差异保存在 migrations\versions 下，当前差异脚本为 9f233265796c.py。

使用数据库迁移工具 flask-migrate 时进行数据迁移，如图 11-6 所示。

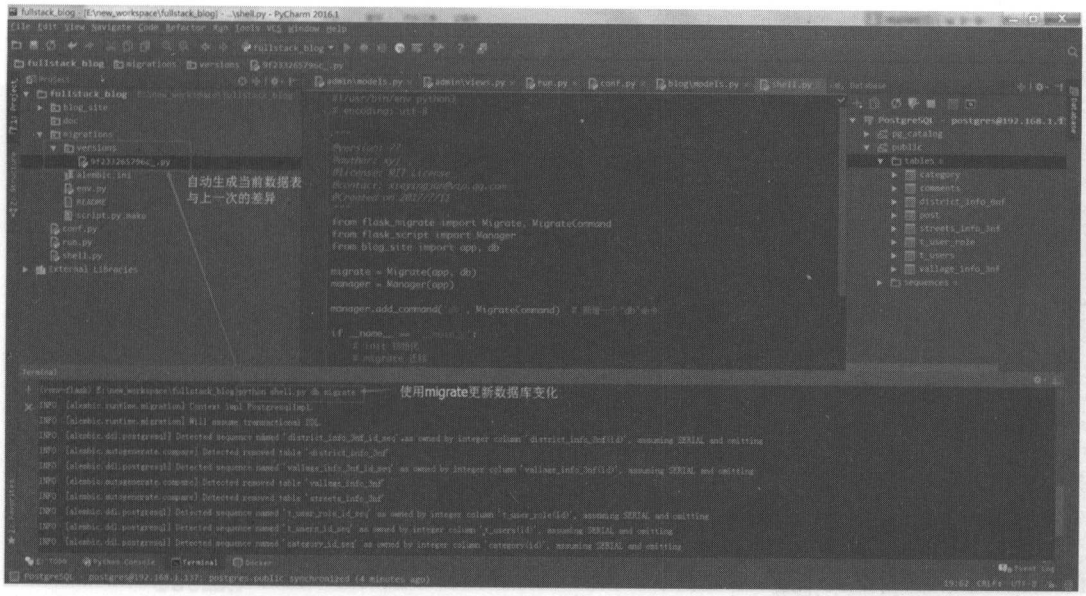


图 11-6

```
>python shell.py db migrate  # 检查数据表变化
```

4. 使用 flask-migrate 升级数据库模型

在控制台下进入虚拟环境，运行 shell.py 脚本，传入命令 db，传入参数 migrate，同步一次数据结构是否有变化，完成后传入参数 upgrade 升级数据模型，如图 11-7 所示。

```
>python shell.py db migrate  # 同步检查数据模式是否有变化
>python shell.py db upgrade  # 将数据库升级到最新数据模型
```

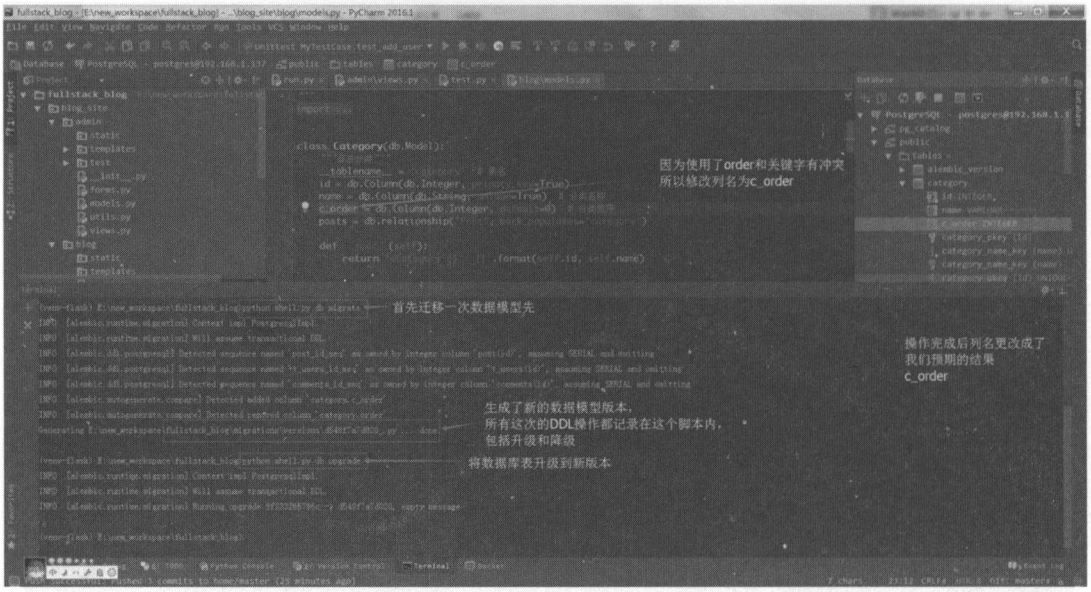


图 11-7

5. 创建后台管理视图

admin/views.py

```
#!/usr/bin/env python3
# encoding: utf-8

"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/11
"""
from flask import Blueprint, render_template, url_for, redirect, request,
flash, g
from flask_login import login_user, logout_user, current_user,
login_required
from blog_site import app, login_manager, db
from blog_site.admin.models import Users, UserRole
from blog_site.admin.forms import UserRoleForm, UsersForm
from blog_site.admin.utils import allowed_file, upload_file
from blog_site.blog.models import Post, Comment, PostForm, Category,
CategoryForm
from datetime import datetime
```

```
# 使用笨办法保存上传文件时引入下面的包
# from werkzeug.utils import secure_filename
# import os

admin = Blueprint('admin', __name__, template_folder='templates',
static_folder='static')  # 创建实例，指定当前文件夹内的模板和静态文件位置


def md5(str):
    """MD5 加密"""
    import hashlib  # 引入内置加密库
    if len(str) != 0:
        m = hashlib.md5()  # 实例化一个 md5 对象
        m.update(str.encode())  # 更新 m 对象，传入需要加密的字符串 str
return m.hexdigest()  # 返回 md5 加密字符串
    else:
        return ""


@login_manager.user_loader
def load_user(userid):
    """
    这个回调用于从会话中存储的用户 ID 重新加载用户对象。
    它应该接受一个用户的 unicode ID 作为参数，并且返回相应的用户对象。
    一般用于"记住我"后 cookies 中记录用户 ID，直接从
    """
    return Users.query.get(int(userid))


@app.before_request
def before_request():
    g.user = current_user  # 放入全局 g 中


@login_manager.request_loader
def request_loader(request):
    """
    有时你可能在不使用 cookies 的情况下登录用户，
    比如使用 HTTP 头或者一个作为查询参数的 API 密钥。
    这种情况下，你应该使用 request_loader 回调。
    这个回调和 user_loader 回调作用一样，
    但是 user_loader 回调只接受 Flask 请求而不是一个 user_id。
```

```
    """
    username = request.form.get('username')  # 从前端获取用户名
    pwd = request.form.get('pwd', '')  # 从前端获取登录密码
    pwd_md5 = md5(pwd)  # 将前端密码转换 md5 序列
    if Users.query.filter(Users.role_id == 1).filter(Users.name ==
username).filter(
                Users.password == pwd_md5).count() > 0:  # 数据库查找记录
        user = Users()  # 实例化一个用户对象
        user.name = username  # 传入用户名
        user.is_authenticated = pwd_md5 == user.password  # 当用户通过验证时,
# 即提供有效证明时返回 True 。
        # (只有通过验证的用户才满足 login_required 的条件。)
        return user
    return


@admin.route('/', methods=['GET', 'POST'])
@login_required
def admin_dashboard():
    """ 登录后首页 """
    users = len(Users.query.all())  # 统计注册人数
    posts = len(Post.query.all())  # 统计日志总数
    comments = len(Comment.query.all())  # 统计留言总数
    cur = [users, posts, comments]
    return render_template('admin/index.html', cur=cur)
#将数据返回前端显示


@admin.route('/posts/')
@admin.route('/posts/page/<int:page>/')
@login_required
def post_list(page=1):
    """ 日志列表 """
    posts = Post.query.order_by('post.id desc').paginate(page=page,
per_page=2)  # 分页显示条数
    return render_template('admin/posts-list.html', posts=posts)


@admin.route('/posts/create/', methods=['GET', 'POST'])
@admin.route('/posts/edit/<int:id>/', methods=['GET', 'POST'])
@login_required
def post_create(id=None):
    if id:
```

```
            post = Post.query.filter(Post.id == id).one()
        else:
            post = Post()
        post.created_time = datetime.now()  # 通过表单元素或对象实例来取具体字段值
        form = PostForm(request.form, obj=post)  # 将对象赋值到表单中
        form.category_id.choices = [(c.id, c.name) for c in Category.query.all()]
        form.created_time.data = datetime.now()  # 使用服务器当前时间
        if form.validate_on_submit():  # 检测提交表单是否通过验证
            form.category_id.data = int(form.category_id.data)
# 将分类值字符转化成整型值
            form.populate_obj(post)  # 将 form 表单值填充到新对象 post 的字段中
            if allowed_file(request.files['topic_img'].filename):
# 上传文件后缀是否符合定义后缀
                # 笨方法保存上传图片
                # filename =
secure_filename(request.files['topic_img'].filename)
# 检测上传文件名是否合法
                # filepath = os.path.join(app.config['UPLOAD_FOLDER'], filename)
# 拼接文件保存路径
                # request.files['topic_img'].save(filepath)  # 保存文件
                # 笨方法保存上传图片
                filename = upload_file(request.files['topic_img'])
# 使用助手类方法保存图片
                post.topic_img = filename  # 将保存图片名称替换 post.topic_img 值
            post.visble = bool(form.visible.data)  # 将上传字符串转换成布尔型
            db.session.add(post)  # 插入日志数据
            db.session.commit()  # 数据库提交记录
            # return str(form.data)  # 排查问题方法，前端直接显示提交内容
            return redirect(url_for('admin.post_list'))
        else:

            return render_template('admin/post-create.html', form=form)


@admin.route('/post/del/<int:id>/', methods=['GET', 'POST'])
@login_required
def post_del(id=None):
    """日志删除"""
    post = db.session.query(Post).filter(Post.id == id).one()
# 通过 id 查找相应记录
    db.session.delete(post)  # 删除记录
    db.session.commit()  # 数据库提交保存
```

```
    return redirect(url_for('admin.post_list'))  # 返回日志列表


@admin.route('/category/')
@admin.route('/category/page/<int:page>/')
@login_required
def post_category(page=1):
    """日志分类"""
    category = Category.query.order_by('category.id').paginate(page=page,
per_page=10)  # 分页显示条数
    return render_template('admin/post-category-list.html',
category=category)


@admin.route('/category/create/', methods=['GET', 'POST'])
@admin.route('/category/edit/<int:id>/', methods=['GET', 'POST'])
@login_required
def post_category_edit(id=None):
    """日志分类编辑"""
    if id:
        area = Category.query.filter(Category.id == id).one()  # 查询到该 ID
# 内容赋值到实例中
    else:
        area = Category()  # 创建新分类
    form = CategoryForm(request.form, obj=area)  # 将对象赋值到表单中
    if form.validate_on_submit():  # 检测提交表单是否通过验证
        form.populate_obj(area)  # 将 form 表单值填充到新对象 area 的字段中
        if id == None:
            db.session.add(area)  # 插入新数据
        db.session.commit()  # 数据库提交保存
        return redirect(url_for('admin.post_category'))
    return render_template('admin/post-category-edit.html', form=form)


@admin.route('/category/del/<int:id>/', methods=['GET', 'POST'])
@login_required
def post_category_del(id=None):
    """日志分类删除"""
    postcategory = db.session.query(Category).filter(Category.id ==
id).one()  # 查找对应记录
    db.session.delete(postcategory)  # 删除记录
    db.session.commit()  # 数据库提交保存
    return redirect(url_for('admin.post_category'))
```

```
@admin.route('/comments/')
@admin.route('/comments/page/<int:page>/')
@login_required
def comments_list(page=1):
    """ 留言列表 """
    comments = Comment.query.order_by('comments.id').paginate(page=page,
per_page=10)
    return render_template('admin/comments-list.html', comments=comments)


@admin.route('/user-group/')
@admin.route('/user-group/<int:page>/')
@login_required
def user_group(page=1):
    """用户分组"""
    user_group_list =
UserRole.query.order_by('t_user_role.id').paginate(page=page,
per_page=20)
    return render_template('admin/user-group.html',
user_group_list=user_group_list)


@admin.route('/user-group/create/', methods=['GET', 'POST'])
@admin.route('/user-group/edit/<int:id>/', methods=['GET', 'POST'])
@login_required
def user_group_edit(id=None):
    """用户分组编辑"""
    if id:
        user_group = UserRole.query.filter(UserRole.id == id).one()
# 查找记录
    else:
        user_group = UserRole() # 实例化新对象
    form = UserRoleForm(request.form, obj=user_group)  # 将对象赋值到表单中
    if form.validate_on_submit():  # 检测提交表单是否通过验证
        form.populate_obj(user_group)
# 将 form 表单值填充到新对象 user_group 的字段中
        if id == None:  # 判断新建还是更改数据
            db.session.add(user_group)  # 查询新数据
        db.session.commit()  # 数据库提交
        return redirect(url_for('admin.user_group'))
    else:
```

```
    return render_template('admin/user-group-edit.html', form=form)


@admin.route('/user-group/delete/<int:id>/')
@login_required
def user_group_delete(id):
    """用户分组删除"""
    user_group_id = db.session.query(UserRole).filter(UserRole.id ==
id).one()  # 查找到记录
    db.session.delete(user_group_id)  # 删除记录
    db.session.commit()  # 数据库提交保存
    return redirect(url_for('admin.user_group'))


@admin.route('/user-list/')
@admin.route('/user-list/<int:page>/')
@login_required
def user_list(page=1):
    """用户信息列表"""
    users_list = Users.query.order_by('t_users.id').paginate(page=page,
per_page=9)
    return render_template('admin/user-list.html', users_list=users_list)


@admin.route('/user-list/edit/', methods=['GET', 'POST'])
@admin.route('/user-list/edit/<int:id>/', methods=['GET', 'POST'])
@login_required
def users_edit(id=None):
    """ 修改和新增用户 """
    if id:  # 判断是否有 id 传入
        users = Users.query.filter(Users.id == id).one()
# 如果有则查询对应 id 的用户资料
    else:
        users = Users()  # 否则实例化一个新用户
    form = UsersForm(request.form, obj=users)  # 将对象赋值到表单中
    form.role_id.choices = [(c.id, c.role) for c in UserRole.query.all()]
    if request.method == 'POST':
        form.role_id.data = int(form.role_id.data)  # 将传入的用户权限字符转成
#整型
        if form.password.data != users.password:
# 判断传入的密码是否等于传出的密码
            form.password.data = md5(form.password.data)  # md5 加密传入的密码
        form.populate_obj(users)  # 将 form 表单值填充到新对象 users 的字段中
```

```
        if id == None:
            db.session.add(users)  # 如果传入 id 为空则添加用户
        db.session.commit()  # 提交保存
        return redirect(url_for('admin.user_list'))  # 返回用户列表
    else:
        return render_template('admin/users-edit.html', form=form)


@admin.route('/user-list/delete/<int:id>/')
@login_required
def users_delete(id):
    """ 删除用户信息 """
    users_id = db.session.query(Users).filter(Users.id == id).one()
    db.session.delete(users_id)
    db.session.commit()
    return redirect(url_for('admin.user_list'))


@admin.route('/login/', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form.get('username', None)  # 从 post 提交中获取表
#单 username 字段
        pwd = request.form.get('pwd', None)  # 从 post 提交中获取表单 pwd 字段
        pwd_md5 = md5(pwd)  # 将前端传过来的 pwd 转换成 md5，安全性欠佳，最好把前端
#直接加密后传送过来
        registered_user = Users.query.filter(Users.role_id ==
1).filter(Users.name == username).filter(
            Users.password == pwd_md5).first()  # 查找用户权限等于 1 且用户名和密
#码有返回第一条数据
        # 如同 select * from t_users where role_id =1 and name=username and
password = pwd
        if registered_user:  # 如果有则返回并执行下面语句
            user = Users()  # 实例化用户
            user.id = registered_user.id  # 用户 id 等于查询结果中的 id
            login_user(user)  # 将用户保存到会话状态中
            return redirect(url_for('admin.admin_dashboard'))
        else:
            flash('用户名或密码错误', '错误')  # 返回闪现消息
    return render_template('admin/login.html')


@admin.route('/logout/')
```

```
@login_required
def logout():
    logout_user()  # 清除用户信息
    return redirect(url_for('admin.login'))
```

说明：上面查询使用了.one()去获取单条数据和.all()去获取全部数据。在使用过程中，预知结果是只返回一条数据，使用.one()比使用.all()高效。因为数据库不知道要返回多少条数据，当指定只返回一条数据的时候，数据库查询到第一条记录就会自动结束查询，而使用.all()查询，数据库则在跑完全部记录后才返回结果。

6. 创建 HTML 基类模板

前面创建好的响应视图，需要配合 HTML 模板才能有更好的显示效果。这里我们使用 jinjia 语法创建一个 layout.html 基页，其他页面则通过继承这个基页来展现。

```
admin/templates/admin/layout.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>管理员信息列表</title>
{#    引入样式#}
    <link rel="stylesheet" type="text/css"
href="http://cdn.bootcss.com/bootstrap/3.3.6/css/bootstrap.min.css">
    <link rel="stylesheet" type="text/css"
href="http://cdn.bootcss.com/font-awesome/4.6.2/css/font-awesome.min.cs
s">
{#    浏览器自适应#}
    <meta name="viewport" content="width=device-width,initial-scale=1">
{#    自定义居中样式#}
    <style type="text/css">
        .value {
            text-align: center;
        }
    </style>
</head>
<body>
<!-- 导航栏 -->
<nav role="navigation" class="navbar navbar-default navbar-static-top">
    <div class="container">
        <div class="row">
            <!-- 面包菜单 -->
```

```
            <div class="navbar-header navar-right">
                <button data-target="#navbarCollapse"
data-toggle="collapse" class="navbar-toggle">
                    <span class="sr-only">切换菜单</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a href="#" class="navbar-brand"><b>后台管理</b></a>
            </div>
            <div id="navbarCollapse" class="collapse navbar-collapse">
                <ul class="nav navbar-nav">
                    <li {{ 'class=active' if nav_active =='index' else '' }}><a
                            href="{{ url_for('admin.admin_dashboard') }}">首
                                页</a></li>
                    <li {{ 'class=active' if nav_active =='category' else
                            '' }}><a
                            href="{{ url_for('admin.post_category') }}">日志分
类管理</a></li>
                    <li {{ 'class=active' if nav_active =='post' else '' }}><a
href="{{ url_for('admin.post_list') }}">日志管理</a>
                    </li>
                    <li {{ 'class=active' if nav_active =='comment' else
                            '' }}><a
                            href="{{ url_for('admin.comments_list') }}">留言管
                                理</a></li>
                    <!-- 下拉菜单 -->
                    <li {{ 'class=active' if nav_active =='user' else '' }}
role="presentation" class="dropdown">
                        <a class="dropdown-toggle" data-toggle="dropdown"
href="#" role="button" aria-haspopup="true"
                           aria-expanded="false">
                            用户管理 <span class="caret"></span>
                        </a>
                        <ul class="dropdown-menu">
                            <li><a href="{{ url_for('admin.user_group') }}">
用户分组</a></li>
                            <li><a href="{{ url_for('admin.user_list') }}">用
户信息</a></li>
                        </ul>
                    </li>
                </ul>
                <form class="nav navbar-form navbar-left">
```

```
                <div class="form-group">
                    <div class="input-group">
                        <input type="text" class="form-control"
name="search" placeholder="搜索...">
                    </div>
                </div>
                <button type="submit" class="btn btn-default">搜索
</button>
            </form>
            <div class="pull-right">
                <ul class="nav navbar-nav">
                    <li class="dropdown">
                        <a href="#" class="dropdown-toggle"
data-toggle="dropdown" role="button"
                           aria-haspopup="true"
                           aria-expanded="false">欢迎{{ current_user.name
if current_user.is_authenticated else '未登录' }}<span
                                   class="caret"></span></a>
                        <ul class="dropdown-menu">
                            <li><a href="{{ url_for('admin.login') }}">退
出</a></li>
                        </ul>
                    </li>
                </ul>
            </div>
        </div>
    </div>
</nav>
<!-- 展示内容 -->
{% block main_content %}
{% endblock %}
</body>
<!-- 加载 js 脚本 -->
<script src="http://cdn.bootcss.com/jquery/2.2.3/jquery.min.js"></script>
<script
src="http://cdn.bootcss.com/bootstrap/3.3.6/js/bootstrap.min.js"></scri
pt>
{#预留 js 块让其他页面有特殊 js 可以加载#}
{% block js_page %}
{% endblock %}
</html>
```

说明：在模板目录下在创建 admin 目录是为了避免 flask 的 bug，在 0.12.2 的版本里还是有模板引用的 bug，当两个蓝图模板都使用同一个页面名字时，flask 会输出最先找到的页面。即假设 admin/templaes/中有一个 index.html 的页面，而 blog 模板在 templates 目录下又有一个 index.html 的页面，当 blog 视图输出 render_template('index.html')的时候，flask 会显示最快找到的 index.html 的文件，即显示 admin/templates 目录下的 index.html，这与我们预期的输出不符。为了解决这个问题，就要在每个蓝图模板目录下再创建一个目录来保存页面。例如 admin/templaes/admin/index.html，在 admin 视图中返回 render_template('admin/index.html')，在 blog 视图中返回 render_template('blog/index.html')，这样就不会产生这个问题了。

7. 引用基类模板

```
{% extends "layout.html" %}  # 继承基页模板
{% import 'helper.html' as helper %}  # 引入助手类，将分页功能写成助手类有利于
# 代码可重用

{% set nav_active = 'category' %}  # 设置变量 nav_active 传至基页，用于激活当前
# 选择的导航键

{% block main_content %}  # 用下列替换基页中的模块
<div class="row">
  <div class="col-lg-8 col-lg-offset-2 col-md-8 col-md-offset-2 col-sm-8
col-sm-offset-2">
      <section class="panel">
          <header class="panel-heading head-border">
              日志分类
          <a href="{{ url_for('admin.post_category_edit') }}" class="btn
btn-primary btn-xs" title="新增"><i class="fa fa-pencil"></i></a>
          </header>
          <table class="table table-hover">
              <thead>
              <tr>
                  <th>#</th>
                  <th>分类</th>
                  <th>排序</th>
              </tr>
              </thead>
              <tbody>
              {% for item in category.items %}
              <tr>
                  <td>{{item.id}}</td>
```

```
                    <td>{{item.name}}</td>
                    <td>
                        <a
href="{{ url_for('admin.post_category_edit',id=item.id) }}" class="btn
btn-primary btn-xs" title="编辑"><i class="fa fa-pencil"></i></a>
                        <a
href="{{ url_for('admin.post_category_del',id=item.id) }}" class="btn
btn-danger btn-xs" title="删除"><i class="fa fa-trash-o "></i></a>
                    </td>
                </tr>
                {% endfor %}
                </tbody>
            </table>
            <div class="text-center">

{{ helper.render_pagination(category,'admin.post_category') }}  # 分页功能
            </div>
        </section>
    </div>
</div>
{% endblock %}  # 代码块结束标志
```

模板继承关系如图 11-8 所示。

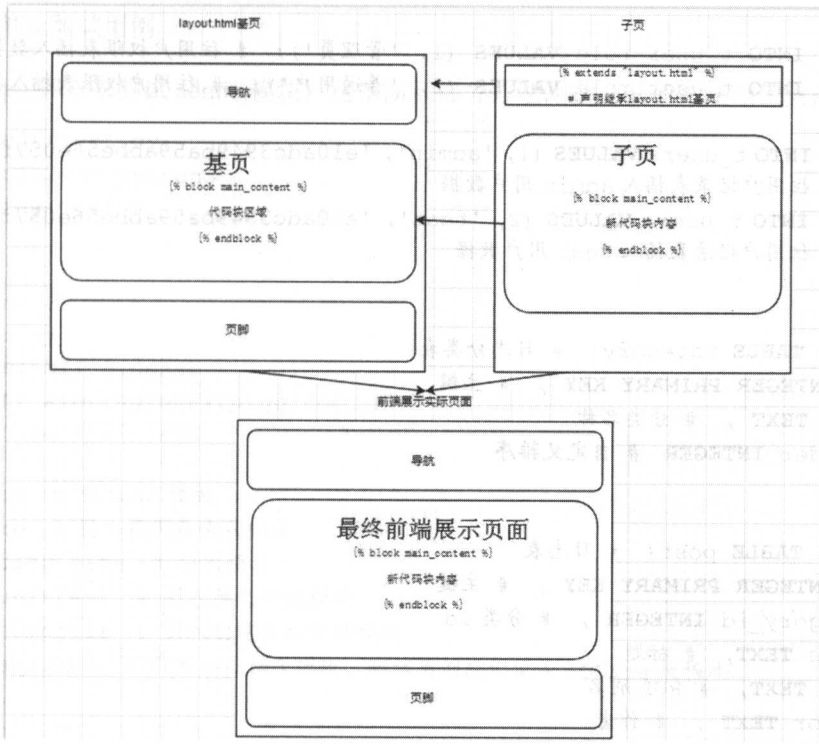


图 11-8

8. 单元测试准备工作

在 admin 模块目录下创建 test 文件夹用来保存测试所需脚本。先创建一个 SQL 脚本，在测试的时候创建临时数据库。

```
admin/test/creat_table.sql:
CREATE TABLE t_user_role(  # 创建用户权限表
  id INTEGER PRIMARY KEY ,  # 主键
  role TEXT  # 权限名称
);

CREATE TABLE t_users(  # 创建用户记录表
  id INTEGER PRIMARY KEY ,  # 主键
  name TEXT ,  # 用户名
  password TEXT ,# 用户密码
  role_id INTEGER ,  # 用户权限
  FOREIGN KEY(role_id) REFERENCES t_user_role(id)  # 外键关联用户权限表
);
```

```
INSERT INTO t_user_role VALUES (1, '管理员'); # 往用户权限表插入数据
INSERT INTO t_user_role VALUES (2, '普通用户'); # 往用户权限表插入数据

INSERT INTO t_users VALUES (1, 'admin', 'e10adc3949ba59abbe56e057f20f883e',
1); # 往用户记录表插入 admin 用户数据
INSERT INTO t_users VALUES (2, 'test', 'e10adc3949ba59abbe56e057f20f883e',
2); # 往用户记录表插入 test 用户数据


CREATE TABLE category( # 日志分类表
  id INTEGER PRIMARY KEY , # 主键
  name TEXT , # 分类名称
  c_order INTEGER # 自定义排序
);

CREATE TABLE post( # 日志表
  id INTEGER PRIMARY KEY , # 主键
  category_id INTEGER , # 分类 id
  title TEXT, # 标题
  slug TEXT, # RUL 别名
  author TEXT , # 作者
  summary TEXT , # 摘要
  topic_img TEXT , # 主题图
  body TEXT , # 正文
  tags TEXT , # 标签
  hits INTEGER , # 点击数
  visble INTEGER , # 是否显示
  created_time TEXT , # 日志创建时间
  FOREIGN KEY(category_id) REFERENCES category(id) # 外键关联日志分类表
);

CREATE TABLE comments( # 留言表
  id INTEGER PRIMARY KEY , # 主键
  post_id INTEGER , # 日志 id
  name TEXT , # 用户名
  text TEXT , # 留言内容
  date TEXT , # 留言时间
  FOREIGN KEY(post_id) REFERENCES post(id) # 外键关联日志表
);
```

9. 创建单元测试用例

创建 test.py 脚本来测试 admin 模块，这里演示部分方法测试，其他方法测试大同小异。

admin/test/test.py:

```
#!/usr/bin/env python3
# encoding: utf-8

"""
@version: ??
@author: xyj
@license: MIT License
@contact: xieyingjun@vip.qq.com
@Created on 2017/7/12
"""
import os  # 引入 os 模块
import re  # 引入正则表达式模块
import blog_site  # 引入项目
import unittest  # 引入单元测试模块
import tempfile  # 引入创建临时文件模块
import sqlite3  # 引入 sqlit3 模块，测试用例我们使用 SQLite 数据库


class MyTestCase(unittest.TestCase):
    def setUp(self):
        """ 测试开始前准备工作 """
        self.db_pathd, self.db_path = tempfile.mkstemp(suffix=".db")  # 创
#建临时数据文件
        # print(self.db_path)  # 打印临时数据文件存放路径
        blog_site.app.config['SQLALCHEMY_DATABASE_URI'] =
'sqlite:///{}'.format(self.db_path.replace('\\', '/'))  # 将项目的数据连接更
#改成临时文件文件路径
        blog_site.app.config['TESTING'] = True  # 标记测试为真
        self.app = blog_site.app.test_client()  # 实例化测试客户端
        self.init_db(self.db_path)  # 初始化数据库

    def tearDown(self):
        """ 测试完成后清理工作 """
        os.close(self.db_pathd)  # 关闭数据文件句柄
        os.unlink(self.db_path)  # 删除数据文件

    def init_db(self, database):
        """ 初始化数据库往数据库内写数据 """
```

```
        db_path = database.replace('\\', '/')
        # print(db_path)  # 打印数据文件路径
        db = sqlite3.connect(db_path)  # 连接数据文件
        c = db.cursor()  # 创建光标
        with open('creat_table.sql', 'r', encoding='utf-8') as f:  # 读取遍
#历 SQL 脚本
            c.executescript(f.read())  # 批量执行脚本内语句
        db.commit()  # 提交数据库

    def login(self, username, password):  # 登录方法
        return self.app.post('/admin/login/', data=dict(  # 使用 post 方法构
# 建一个字典表
            username=username,  # 传入用户名
            pwd=password  # 传入用户密码
        ), follow_redirects=True)  # 参数 follow_redirects 为重定向追踪

    def test_login(self):
        """ 测试登录界面 """
        # 成功登录
        re_data = self.login('admin', '123456')  # 传递正确用户名和密码
        print(re_data.data) # 打印页面返回信息
        assert '欢迎 admin' in re_data.data.decode() # 断言"欢迎 admin"语句在
# 返回页面中
        # 用户无权限
        re_data = self.login('test', '123456')  # 传递正确用户名和密码但无访问
#权限
        assert '用户名或密码错误' in re_data.data.decode()  # 断言语句是否在返回
# 记录中
        # 用户密码错误
        re_data = self.login('admin', '123')  # 传递正确用户名错误密码
        assert '用户名或密码错误' in re_data.data.decode() # 断言句是否在返回记
# 录中

    def logout(self):  # 退出方法
        return self.app.get('/admin/logout/', follow_redirects=True)  # get
#访问登出连接

    def test_logout(self):
        """ 测试退出 """
        # 退出登录
        re_data = self.logout()  # 执行登出
        assert '后台登录' in re_data.data.decode()  # 断言返回页面是否在登录页面
```

```
        # 再次访问需授权页面
        re_data = self.app.get('/admin/user-list', follow_redirects=True)
# 再次访问需授权才能访问页面
        assert '请登录' in re_data.data.decode()  # 断言提示'请登录'是否在放回
# 内容中

    def test_add_user(self):
        """ 创建新用户 """
        # 管理员登录
        re_data = self.login('admin', '123456')
        assert '欢迎 admin' in re_data.data.decode()
        # 访问用户列表页
        re_data = self.app.get('/admin/user-list/', follow_redirects=True)
        assert 'test' in re_data.data.decode()
        # get 访问新增用户
        re_data = self.app.get('/admin/user-list/edit/',
follow_redirects=True)
        token = re.findall(r'csrf_token.*value="(.*)"',
re_data.data.decode())  # 使用正则获取表单中的 csrf_token 值
        # post 提交新用户资料
        rv = self.app.post('/admin/user-list/edit/', data=dict(
            csrf_token='{}'.format(token[0]),
            name='test2',
            password='123456',
            role_id='1'
        ), follow_redirects=True)
        print(rv.data.decode())
        assert 'test2' in rv.data.decode()  # 断言是否创建 test2 用户成功


if __name__ == '__main__':
    unittest.main()  # 执行单元测试
```

通过上面的代码，很容易发现很多代码的核心都是围绕数据的增删改查操作，在核心外围包裹着授权、数据规整等操作。看透了这些本质之后，根据自己的需求就能定制出符合自己业务需求的后台。具体项目代码请从 https://gitlab.com/lanmaokafei/fullstack_blog 下载。

10. 查看项目构建

在项目 pipelines 中可以看到最后构建的内容，如图 11-9 所示。

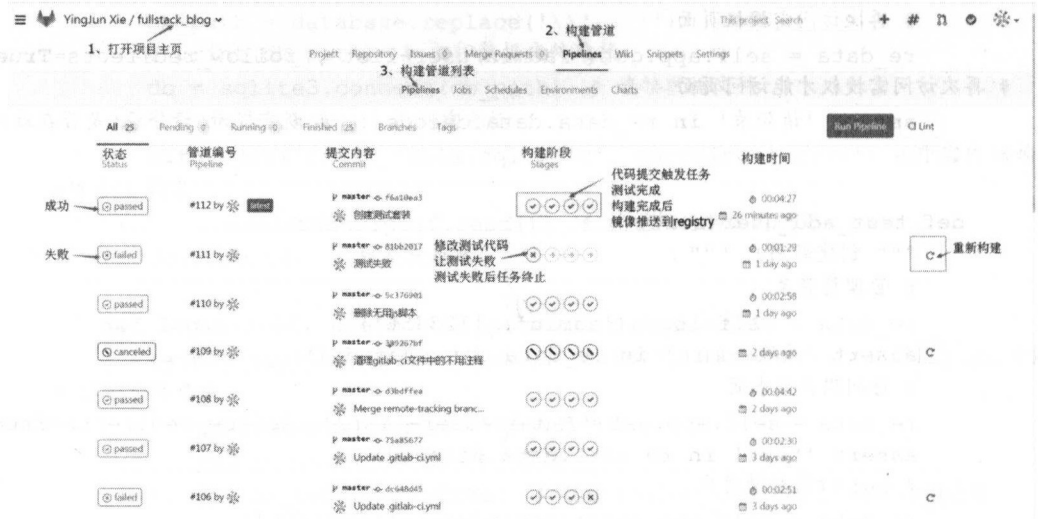


图 11-9

11. 从 registry 仓库中 pull 镜像运行

由于在本地演示，所以需要先清除本地构建的镜像，如果镜像在其他服务器，则可以直接略过这一步，如图 11-10 所示。

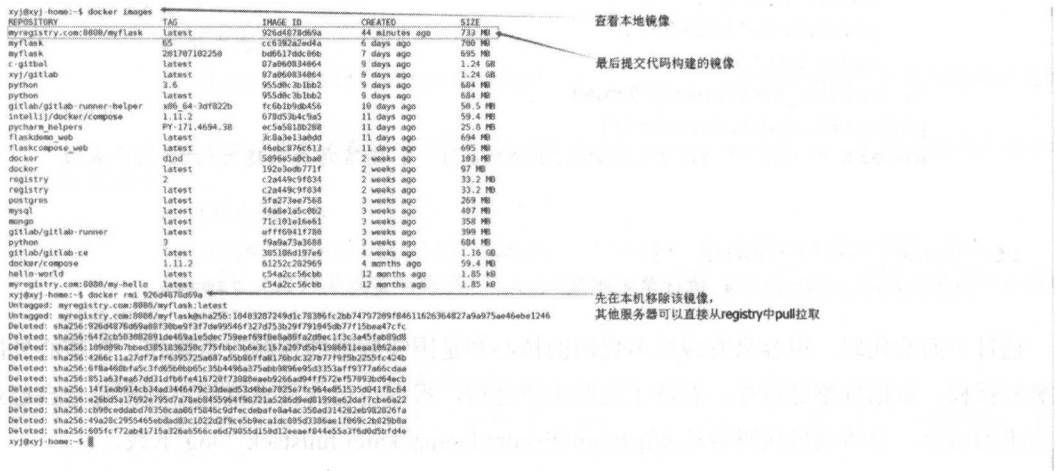


图 11-10

从 registry 服务器中拉取构建镜像，如图 11-11 所示。

```
$ docker pull myregistry.com:8080/myflask  # 从本地 registry 服务器中拉取镜像
```

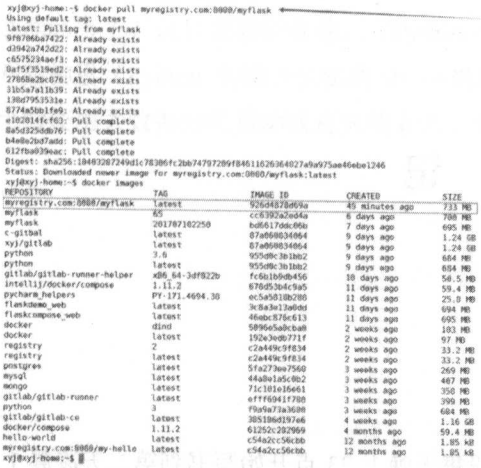


图 11-11

12. 运行镜像

运行镜像并访问，如图 11-12 所示。

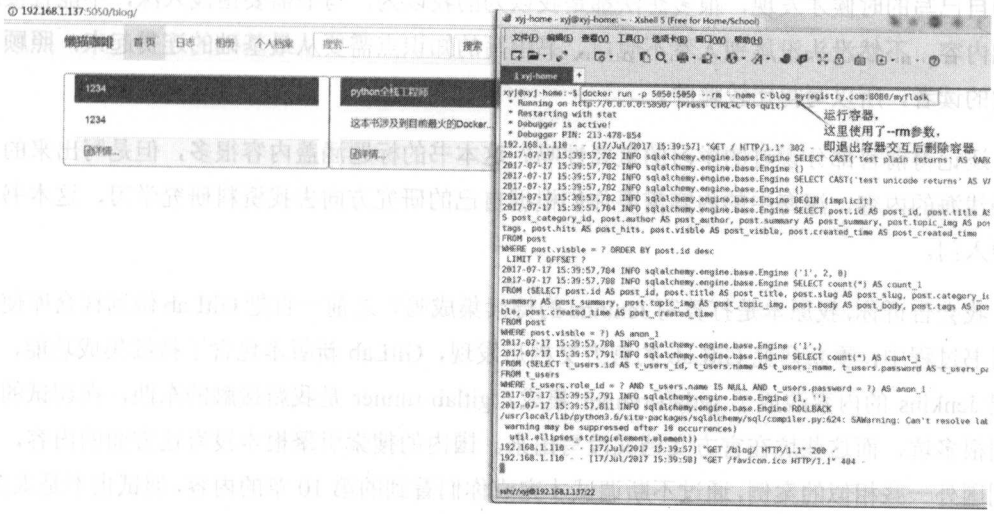


图 11-12

这样就交付镜像给运维去部署在正式环境了。

后 记

不知不觉这本书写了将近 8 个月之久，我一般每天晚上 23 点开始写书到第二天凌晨 1 点。非常感谢我的妻子，她为了不影响我休息，儿子半夜哭闹，都是她自己一个照顾儿子。感谢家人的关心支持，每天夜晚老爸都会大声说“夜麻麻仲唔瞓觉，听日唔使返工呀”。

现在回首，8 个月前看别人写的书，总有一种感觉就是“别人写得太臃肿、太浅显”，而轮到自己写的时候才发现，很多想法都是我以为的我以为，写书需要由浅入深，不能直接跳过基础内容，不然没头没尾别人看不明白。书中写的知识点需要从最基础的连贯起来，照顾不同层次的读者，所以写得比较臃肿。

还记得前言部分说过的学习的重要性吗？这本书的标题涵盖内容很多，但是写出来的都是一些浅海的内容，深海中的内容需要你们根据自己的研究方向去找资料研究学习，这本书只是引导入门。

我会告诉你，我原本是打算写 Jenkins 的持续集成吗？之前一直把 GitLab 做远程仓库使用，在写书过程中，重新研究 GitLab 的官方文档后发现，GitLab 新版本包含了持续集成功能，所以我把 Jenkins 的内容改成了 gitlab-runner 集成，gitlab-runner 是我新接触的东西，在调试的时候遇到很多坑，而这些坑在官方文档又没写出来。国内的搜索引擎根本没有这方面的内容，最后找到国外一些相似的案例，通过不断调试才完成你们看到的第 10 章的内容，调试也不是太多次，总共加起来 110 次，解决完遇到的集成问题。

第一次写书，很多东西都是第一次尝试，平时自己不是太喜欢写东西，文字功底比较差，希望各位读者多多包涵。

这就结束了？这只是新的开始。《Python 全栈开发实践入门》结束了，未来时间陪陪家人后，开始构思《Python 全栈开发实践 2——机器学习》，希望把工作中用到的一些知识分享一下，比如使用大数据分析帮助找走失的老人、分析小偷在哪，等等，未完待续……

投稿联络：安 娜
微信&QQ：80303489
邮箱：anna@phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Python全栈开发实践入门

全栈工程师不应只会前后端开发，而是应该从开发、测试、部署各个方面都有所掌握的全技能人才。

本书使用了热门的 Docker 容器技术、GitLab 版本控制、GitLab-runner 持续集成、Python Web Flask 框架等，将一整套开发流程通过简单案例展现出来。

本书内容

- 安装 Ubuntu 系统
- Python 开发工具——sublime3 使用
- Python 开发工具——PyCharm 使用
- Python 开发工具——Vim 使用
- Docker 的安装搭建
- Git 使用
- 数据库介绍
- 基于 Flask 开发 Web 项目
- Web 自动化测试
- 持续集成
- 实战开发简易博客后台

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：安　娜
封面设计：吴海燕

上架建议：程序设计

ISBN 978-7-121-32811-4

定价：69.00元